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

Ahmed M. Aly[1], Ahmed R. Mahmood[1], Mohamed S. Hassan[1], Walid G. Aref[1],
Mourad Ouzzani[2], Hazem Elmeleegy[3], Thamir Qadah[1]
[1]Purdue University, West Lafayette, IN
[2]Qatar Computing Research Institute, Doha, Qatar
[3]Turn Inc., Redwood City, CA
[1]{aaly, amahmoo, msaberab, aref, tqadah}@cs.purdue.edu,
[2]mouzzani@qf.org.qa, [3]hazem.elmeleegy@turn.com

ABSTRACT

The unprecedented spread of location-aware devices has resulted in a plethora of location-based services in which huge amounts of spatial data need to be efficiently processed by large-scale computing clusters. Existing cluster-based systems for processing spatial data employ static data-partitioning structures that cannot adapt to data changes, and that are insensitive to the query workload. Hence, these systems are incapable of consistently providing good performance. To close this gap, we present AQWA, an adaptive and query-workload-aware mechanism for partitioning large-scale spatial data. AQWA does not assume prior knowledge of the data distribution or the query workload. Instead, as data is consumed and queries are processed, the data partitions are incrementally updated. With extensive experiments using real spatial data from Twitter, and various workloads of range and k-nearest-neighbor queries, we demonstrate that AQWA can achieve an order of magnitude enhancement in query performance compared to the state-of-the-art systems.

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 geotagged information are created every day. Meanwhile, the MapReduce framework [14] 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 [43], the open-source implementation of MapReduce, e.g., Pig [32], Hive [41], Cheetah [11], and Pigeon [18], 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.

*This research was supported in part by National Science Foundation under Grants IIS 1117766 and IIS 0964639.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 42nd International Conference on Very Large Data Bases, September 5th - September 9th 2016, New Delhi, India.
Proceedings of the VLDB Endowment, Vol. 8, No. 13
Copyright 2015 VLDB Endowment 2150-8097/15/09.

Existing cluster-based systems for processing large-scale spatial data employ spatial partitioning methods in order to have some pruning at query time. However, these partitioning methods are *static* and cannot efficiently react to data changes. For instance, SpatialHadoop [17, 19] supports static partitioning schemes (see [16]) to handle large-scale spatial data. To handle a batch of new data in SpatialHadoop, the whole data needs to be repartitioned from scratch, which is quite costly.

In addition to being static, existing cluster-based systems are insensitive to the query workload. As noted in several research efforts, e.g., [13, 33, 42], accounting for the query workload can be quite effective. 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.

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. Unlike existing systems that require recreating the partitions, AQWA incrementally updates the partitioning according to the data changes and the query workload. An important characteristic of AQWA is that it does not presume any knowledge of the data distribution or the query workload. Instead, AQWA applies a lazy mechanism that reorganizes the data as queries are processed.

Traditional spatial index structures try to increase the pruning power at query time by having (almost) unbounded decomposition until the finest granularity of data is reached in each split. In contrast, AQWA keeps a lower bound on the size of each partition that is equal to the data block size in the underlying distributed file system. In the case of HDFS, the reason for this constraint is twofold. First, each file is allocated at least one block (e.g., 128 MB) even if the size of the file is less than the block size in HDFS. In Hadoop, each mapper typically consumes one file. So, if too many small partitions exist, too many short-lived mappers will be launched, taxing the system with the associated setup overhead of these mappers. Second, allowing too many small partitions can be 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 NameNode is a centralized resource in Hadoop that manages the metadata of the files in HDFS, and handles the file requests across the whole cluster. Hence, the NameNode is a critical component that, if overloaded with too many small files, slows down the overall cluster (e.g., see [5, 26, 28, 44]).

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. The cost function integrates both the data distribution and the query workload. AQWA repeatedly tries to minimize the cost of query execution by splitting some partitions. In order to choose the partitions to split and find the best positions to split these partitions according to the cost model, two operations are excessively applied: 1) Finding the number of points in a given region, and 2) Finding the number of queries that overlap a given region. A straightforward approach 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 these challenges, AQWA employs a set of main-memory structures that maintain information about the data distribution as well as the query workload. These main-memory structures along with efficient summarization techniques from [8, 25] enable AQWA to efficiently perform its repartitioning decisions.

AQWA supports spatial range and k-Nearest-Neighbor (kNN, for short) queries. Range queries are relatively easy to process in MapReduce-based platforms because the region (i.e., window) that bounds the answer of a query is predefined, and hence the data partitions that overlap that region can be determined before the query is executed. However, the region that confines the answer of a kNN query is unknown until the query is executed. Existing approaches (e.g., [19]) for processing a kNN query over big spatial data require two rounds of processing in order to guarantee the correctness of evaluation, which implies high latency. AQWA presents a more efficient approach that guarantees the correctness of evaluation through a single round of computation while minimizing the amount of data to be scanned during processing the query. To achieve that, AQWA leverages its main-memory structures to efficiently determine the minimum spatial region that confines the answer of a kNN query.

AQWA can react to different types of query workloads. Whether there is a single hotspot area (i.e., one that receives queries more frequently, e.g., downtown area), or there are multiple simultaneous hotspots, AQWA efficiently determines the minimal set of partitions that need to be split, and accordingly reorganizes the data partitions. Furthermore, when the workload permanently shifts from one (or more) hotspot area to another, AQWA is able to efficiently react to that change and update the partitioning accordingly. To achieve that, AQWA employs a time-fading counting mechanism that alleviates the overhead corresponding to older query-workloads.

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

- We introduce AQWA, a new mechanism for partitioning big spatial data that: 1) can react to data changes, and 2) is query-workload-aware. Instead of recreating the partitions from scratch, AQWA adapts to changes in the data by incrementally updating the data partitions according to the query workload.
- We present a cost-based model that manages the process of repartitioning the data according to the data distribution and the query workload while abiding by the limitations of the underlying distributed file system.
- We present a time-fading mechanism that alleviates the repartitioning overhead corresponding to older query-workloads by assigning lower weights to older queries in the cost model.
- Unlike the state-of-the-art approach (e.g., [19]) that requires two rounds of computation for processing a kNN query on big spatial data, we show how a kNN query can be efficiently answered through a single round of computation while guaranteeing the correctness of evaluation.
- We show that AQWA achieves a query performance gain of one order of magnitude in comparison to the state-of-the-art system [19]. This is demonstrated through a real implementation of AQWA on a Hadoop cluster where Terabytes of real spatial data from Twitter are acquired and various workloads of range and kNN queries are processed.

The rest of this paper proceeds as follows. Section 2 discusses the related work. Section 3 gives an overview of AQWA. Sections 4 and 5 describe how data partitioning and query processing are performed in AQWA. Section 6 explains how concurrency and system failures are handled in AQWA. Section 7 provides an experimental study of the performance of AQWA. Section 8 includes concluding remarks and directions for future work.

2. RELATED WORK

Work related to AQWA can be categorized into four main categories: 1) centralized data indexing, 2) distributed data-partitioning, 3) query-workload-awareness in database systems, and 4) spatial data aggregation and summarization.

In the first category, *centralized indexing*, e.g., B-tree [12], R-tree [24], Quad-tree [39], Interval-tree [10], k-d tree [9], the goal is to split the data in a centralized index that resides in one machine. The structure of the index can have unbounded decomposition until the finest granularity of data is reached in each partition. 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 size of each partition because allowing too many small partitions can be very harmful to the overall health of a computing cluster (e.g., see [5, 26, 28, 44]). Moreover, in Hadoop for instance, too many small partitions lead to too many short-lived mappers that can have high setup overhead. Therefore, AQWA keeps a lower bound on the size of each partition that is equal to the data block size in the underlying distributed file system (e.g., 128 MB in HDFS).

In the second category, *distributed data-partitioning*, e.g., [19, 20, 21, 22, 27, 30, 31], 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., keeping a lower bound on the size of each partition. For instance, the Eagle-Eyed Elephant (E3) framework [20] 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. [17, 19] present SpatialHadoop; a system that processes spatial two-dimensional data using two-dimensional Grids or R-Trees. A similar effort in [27] addresses how to build R-Tree-like indexes in Hadoop for spatial data. [35, 34] 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, all the efforts in this category apply static partitioning mechanisms that are neither adaptive nor query-workload-aware. In other words, the systems in this category do not provide a functionality to efficiently update the data partitions after

a set of data updates is received (i.e., appended). In this case, the partitions for the entire dataset need to rebuilt, which is quite costly and may require the whole system to halt until the new partitions are created.

In the third category, *query-workload-awareness in database systems*, several research efforts have emphasized the importance of taking the query workload into consideration when designing the database and when indexing the data. [13, 33] present query-workload-aware data partitioning mechanisms in distributed shared-nothing platforms. However, these mechanisms support only one-dimensional data. [42] presents an adaptive indexing scheme for continuously moving objects. [15] presents techniques for supporting variable-size disk pages to store spatial data. [4] presents query-adaptive algorithms for building R-trees. Although the techniques in [4, 15, 42] are query-workload-aware, they assume a centralized storage and processing platform.

In the fourth category, *spatial data aggregation and summarization*, several techniques have been proposed for efficiently aggregating spatial data and supporting spatial range-sum queries. In AQWA, computing the cost function requires support of two range-sum queries, namely, 1) counting the number of points in a spatial range, and 2) counting the number of queries that intersect a spatial range. [25] presents the idea of maintaining *prefix sums* in a grid in order to answer range-sum queries of the number of points in a window, in constant time, irrespective of the size of the window of the query or the size of the data. The relative prefix sum [23] and the space-efficient relative prefix sum [37] were proposed to enhance the update cost and the space required to maintain the prefixes. [36] further enhances the idea of prefix sum to support OLAP queries. Generalizing the idea of prefix sums in a grid for counting the number of rectangles (i.e., queries) that intersect a spatial region is a bit challenging due to the problem of duplicate counting of rectangles. Euler histograms [8] were proposed to find the number of rectangles that intersect a given region without duplicates. [7] and [40] employ the basic idea of Euler histograms to estimate the selectivity of spatial joins. AQWA employs the the prefix sum techniques in [25] and a variant of the Euler histogram in [8] to compute its cost function in constant time, and hence efficiently determine its repartitioning decisions.

3. PRELIMINARIES

We consider range and kNN queries over a set, say S, of data points in the two-dimensional space. Our goal is to partition S into a set of 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 process of partitioning the data is guided through a cost model that we explain next.

3.1 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 (i.e., points) 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:

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

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

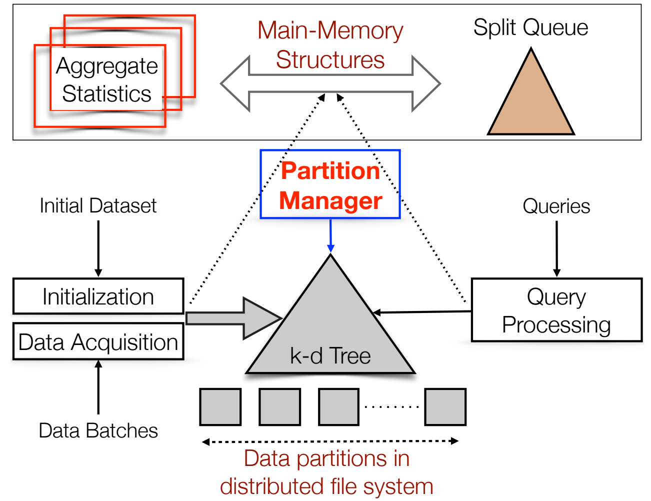


Figure 1: An overview of AQWA.

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. Moreover, to abide by the restrictions of typical distributed file systems, e.g., HDFS, we consider only partitions that are of size greater than a certain limit, e.g., the block size in HDFS. Our goal is to choose the cheapest partitioning layout according to the above equation.

In AQWA, the query workload is not known in advance. As the queries are processed, the query workload should be automatically learned, and the underlying data points should be partitioned accordingly. Similarly, when the query workload changes, the data partitions should be updated accordingly. Furthermore, as data updates are received, the data partitions need to be incrementally updated according to the distribution of the data and query workload.

3.2 Overview of AQWA

Figure 1 gives an overview of AQWA that is composed of two main components: 1) a k-d tree decomposition[1] of the data, where each leaf node is a partition in the distributed file system, and 2) a set of main-memory structures that maintain statistics about the distribution of the data and the queries. To account for system failures, the contents of the main-memory structures are periodically flushed into disk. Upon recovery from a failure, the main-memory structures are reloaded from disk. Four main processes define the interactions among the components of AQWA, namely, Initialization, Query Execution, Data Acquisition, and Repartitioning.

- **Initialization:** This process is performed once. Given an initial dataset, statistics about the data distribution are collected. 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 $O(1)$.

  Based on the counts determined in the Initialization phase, we identify the best partitioning layout that evenly distributes

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

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. We describe the initialization phase in detail in Section 4.1

- **Query Execution:** Given a query, we select 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. After this update, we may (or may not) take a decision to repartition the data.

- **Data Acquisition:** Given a batch of data, we issue a MapReduce job that appends each new data point to its corresponding partition according to the current layout of the partitions. In addition, the counts of points in the grid are incremented according to the corresponding counts in the given batch of data.

- **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.

While the Initialization and Query Execution processes can be implemented in a straightforward way, the Data Acquisition and Repartitioning processes raise the following performance challenges:

- *Overhead of Rewriting:* A batch of data is appended during the Data Acquisition process. To have good pruning power at query time, some partitions need to be split. Furthermore, the overall distribution of the data may change. Thus, we may need to change the partitioning of the data. 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. In Section 7, we show that reconstructing the partitions takes several hours for a few Terabytes of data. This is inefficient especially for dynamic scenarios, where new batches of data are appended on an hourly or daily basis. Hence, we propose an incremental mechanism to alter only a minimal number of partitions according to the query workload.

- *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 maintain main-memory aggregates about the distribution of the data and the query workload. AQWA employs the techniques in [25, 8] to efficiently determine the partitioning decisions via main-memory lookups.

- *Workload Changes and Time-Fading Weights:* AQWA should respond to permanent changes in the query workload. However, we need to ensure that AQWA is resilient to temporary query workloads, i.e., avoid unnecessary repartitioning of the data.

  AQWA keeps the history of all queries that have been processed. However, we need to differentiate between fresh queries, i.e., those that belong to the current query-workload, and relatively old queries. AQWA should alleviate the redundant repartitioning overhead corresponding to older query-workloads. Hence, we apply time-fading weights for the queries in order to alleviate the cost corresponding to old queries according to Equation 1.

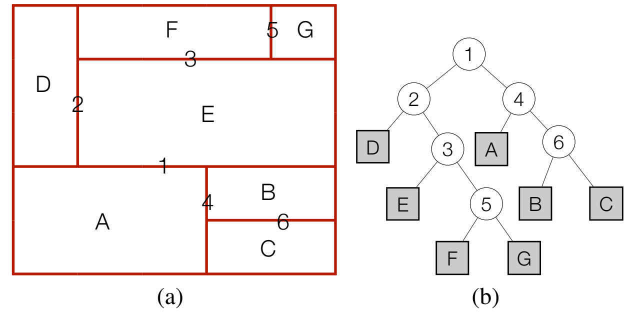


Figure 2: An example tree with 7 leaf partitions.

- *Keeping a lower bound on the size of each partition:* For practical considerations, it is important to avoid small partitions that can introduce a performance bottleneck in a distributed file system (e.g., see [5, 26, 28, 44]). Hence, in AQWA, we avoid splitting a partition if any of the resulting two partitions is of size less than the block size in the distributed file system (e.g., 128 MB in HDFS).

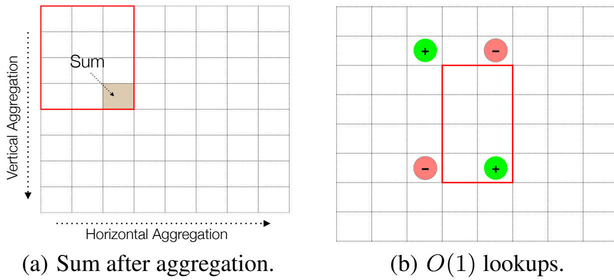
4. AQWA

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 process, we partition the data in a way that balances the number of points across the partitions. In particular, we apply a recursive k-d tree decomposition [9].

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, and stops if any of the resulting partitions is of size $<$ the block size. Figure 2 gives the initial state of an example k-d tree with 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, 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 3.2, 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.



(a) Sum after aggregation.

(b) $O(1)$ lookups.

Figure 3: The prefix-sums for finding the number of points in an arbitrary rectangle in constant time.

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. Hence, in the initialization phase, we preprocess the raw data as in a way that enables quick lookup (in $O(1)$ time) of the count corresponding to a given rectangle. We maintain a 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 that are inside the boundaries of $G[i,\ j]$. 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 using prefix-sums as in [25].

Refer to Figure 3 for illustration. For every row in G, we perform horizontal aggregation, i.e., scan the cells from column 0 to column m and aggregate the values as: $G[i,\ j] = G[i,\ j] + G[i,\ j-1]\ \forall\ j \in [2,\ m]$. Afterwards, we perform vertical aggregation for each column, i.e., $G[i,\ j] = G[i,\ j] + G[i-1,\ j]\ \forall\ i \in [2,\ n]$. 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(a) 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, only four values need to be added/subtracted as shown in Figure 3(b). Thus, the process of finding the number of points for any given rectangle is performed in $O(1)$.

In addition to the above summarization technique, 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, 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 process 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 Data Acquisition and Repartitioning

4.2.1 Data Acquisition

Most sources of big spatial data are dynamic, where new batches of data are received on an hourly or a daily basis. For instance, since 2013, more than 500 Million tweets are created every day [3]. AQWA provides a functionality to append the data partitions with new batches of data. In AQWA, after the Initialization process, each of the partitions in the initial layout is appended with a new set of data points during the Data Acquisition process. Appending a batch of data is performed through a MapReduce job. In the Map phase, each data point is assigned to the corresponding partition, and in the Reduce phase, the data points are appended. In addition, each reducer determines the count of points it receives for the grid cells that overlap with its corresponding partition. A temporary grid receives the aggregate counts and pre-computations are performed the same way we explained in Section 4.1. Afterwards, the counts in the fine-grained grid are incremented with the values in the temporary grid.

We observe that after the data is appended, some (if not all) partitions may increase in size by acquiring more data points. In order to have good pruning at query time, these partitions need to be split. A straightforward approach for altering the partitions is to aggressively repartition the entire data. However this would be quite costly because the process of rereading and rewriting the entire data is prohibitively expensive due to the size of the data. Furthermore, this approach may require the entire system to halt until the new partitions are created. As we demonstrate in Section 7, the process of reconstructing the partitions takes several hours for a few Terabytes of data, which is impractical for dynamic scenarios, e.g., Twitter datasets, where new batches of data need to be appended frequently.

4.2.2 Adaptivity in AQWA

In AQWA, we apply an incremental mechanism that avoids rereading and rewriting the entire dataset, but rather splits a minimal number of partitions according to the query workload. In particular, 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. 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 overhead of reading and writing the contents of these partitions, and the sizes of the resulting 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 4(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 4(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 4(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. Similarly, Partition A is split to A_1 and A_2, and then Partition A_1 is further split to A_{11} and A_{12}. 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:

$$C_d(Split,\ p,\ p_1,\ p_2) = C(p) - C(p_1) - C(p_2). \quad (2)$$

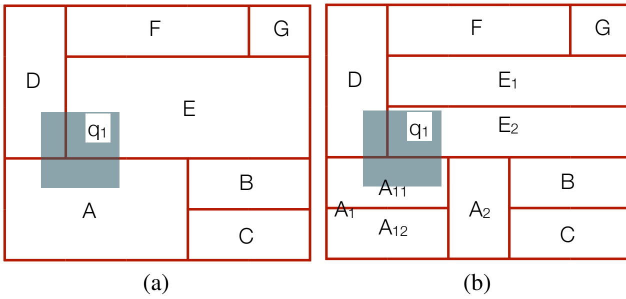


Figure 4: Incremental repartitioning of the data. Only partitions A and E are split.

2. *The cost of read/write during the split operation:* Should we decide to split a partition, the entire data of that partition 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:

$$C_{rw}(p) = 2 \times N(p), \quad (3)$$

where $N(p)$ is the number of points in p.

3. *The size of the resulting partitions:* In AQWA, we ensure that the size of each partition is greater than the block size in the underlying distributed file system. If the number of points in any of the resulting partitions is less than a threshold $minCount$, the split operation is cancelled. $minCount$ is calculated as $\frac{block\ size}{NB}$, where NB is the expected number of bytes that a data point consumes, that can be estimated as the total number of bytes in the dataset divided by the number of points in the dataset.

According to Equations 2 and 3, we decide to split a partition, say p_s if:

$$\begin{aligned} & C_d(Split,\ p_s,\ p_{s1},\ p_{s2}) > C_{rw} \\ & \textbf{and } N(p_{s1}) > minCount \\ & \textbf{and } N(p_{s2}) > minCount. \end{aligned} \quad (4)$$

Observe that 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) change. Two challenges exist in this situation:

1. How can we efficiently determine the best partitions to be split? We need to choose the partition that, if split, would reduce the cost the most.

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. the data distribution only, i.e., without considering the query workload.

To address the first challenge above, i.e., selecting the best partitions to be split, we maintain a priority queue of candidate partitions which we refer to as the split-queue. 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 $C_d - 2 \times N(p_s)$.

After a query is executed, the overlapping partitions are determined and their corresponding values are updated in the priority queue. Observe that if the number of points in any of the resulting partitions of a split operation is $< minCount$, the corresponding partition is not inserted into the split-queue.

4.2.3 Efficient Search for the Best Split

As illustrated in Section 3.1, 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 2, the best split of a partition that would result in the highest cost gain. To compute 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 apply the Euler Histogram that is introduced in [8, 7, 40] to address the problem of duplicate rectangle counting. We extend the fine-grained grid G that maintains counts for the number of points (as explained in Section 4.1). At each grid cell, say $G[i,\ j]$ four additional counters are maintained, 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),

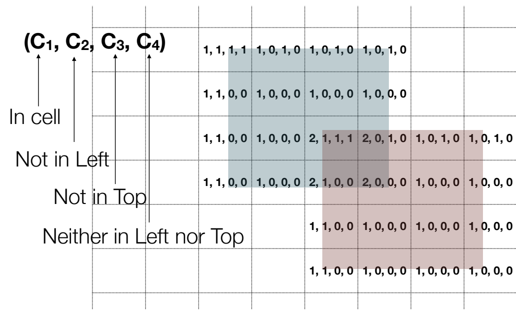


Figure 5: The four counters maintained for each grid cell.

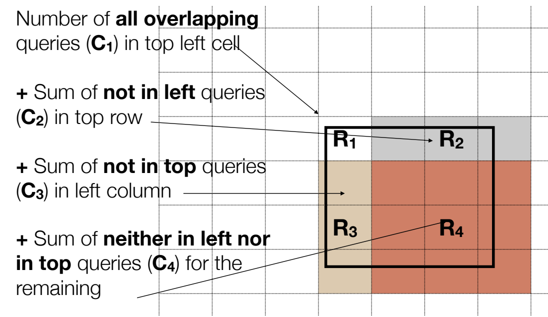


Figure 6: Determining the number of queries that overlap a partition in $O(1)$.

- 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 left nor in top).

Figure 5 gives an illustration of the values of the four counters that correspond to two range queries.

We maintain prefix-sums as in [25] for the values of C_1 through C_4 in a way similar to the one discussed in Section 4.1. For C_2, we maintain horizontal prefix-sums at each row. For C_3, we maintain vertical prefix-sums at each column. 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. Observe that the process of updating the prefix-sums is repeated per query. [23] introduces some techniques for minimizing the update overhead required to maintain the prefix-sums. However, because the fine-grained grid that maintains the counts resides in main-memory it is is cheap to update even if the techniques in [23] are not applied.

A partition, say p, can be divided into four regions R_1 through R_4 as illustrated in Figure 6. To determine the number of queries that overlap a certain partition, say p, we perform the following four operations.

- We determine the value of C_1 in Region R_1, which is the top-left grid cell that overlaps p.
- We determine the aggregate value of C_2 in Region R_2, which is the top border of p except for the top-left cell.
- We determine the aggregate value of C_3 in Region R_3, which is the left border of p except for the top-left cell.
- We determine the aggregate value of C_4 for Region R_4, which is every grid cell that overlaps p except for the left and top borders.

Because prefix-sums are maintained for Counters C_1 through C_4, each of the above aggregate values can be obtained in constant time as in [25]. The sum of the above values represents the number of queries that overlap p. 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.

4.2.4 Accounting for Workload Changes

AQWA is resistant to abrupt or temporary changes to the query workload. Inequality 4 ensures that AQWA does not aggressively split partitions that do not receive queries or that receive queries with lower frequency. A partition is split only if it is worth splitting, i.e., the gain corresponding to the split is greater than the overhead of the split operation. However, when the query workload permanently changes (e.g., moves to a new hotspot area), the number of queries received in the new workload would satisfy Inequality 4, and the desired splits will be granted.

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 4.2.3). 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}. If a new batch of data is received, it may trigger a set of split operations at A_{old}. However, these splits are redundant, i.e., would not lead to any performance gains because the workload has permanently shifted.

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 that we refer to as the *time-fading cycle*. In Section 7.2.2, we study the effect of varying T.

For each of the four counters (refer to c_1 through c_4 in Section 4.2.3) 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 $c > 1$. 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. This requires revisiting each partition to determine its new weight and its new order in the split-queue. A straightforward ap-

proach is to update the values corresponding to all the partitions and reconstruct the split-queue every T time units. However, this approach can be costly because it requires massive operations to rebuild the split-queue. 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 N_p is the number of partitions, we process only $\frac{N_p}{T}$ partitions every time unit. For each of the $\frac{N_p}{T}$ partitions, we recalculate the cost and reinsert these partitions into the split-queue. Eventually, after T time units, all the entries in the split-queue are updated.

5. SUPPORT FOR KNN 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 kNN 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 kNN query depends on the value of k, the location of the query focal point, and the distribution of the data (see [6]). To illustrate, consider the example in Figure 7. 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-closest neighbors of q_2 because of the location of q_2 w.r.t. Partition p (i.e., being near one corner).

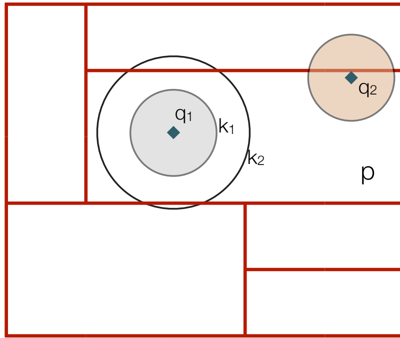


Figure 7: The partitions that contain the k-nearest-neighbors of a query point vary according to the value of k, the location of the query focal point, and the distribution of the data.

[19] 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 kth 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 kNN query. In particular, we make use of the fine-grained virtual grid that contains statistics about the data distribution. Given a kNN query, we determine the grid cells that are guaranteed to contain the answer of the query using the MINDIST and MAXDIST metrics as in [38]. 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 grid cell is greater than M. To illustrate, consider the example in Figure 8. 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.

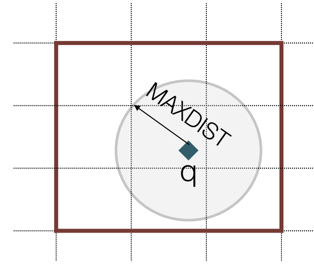


Figure 8: Finding the grid cells (with fine granularity) that are guaranteed to enclose the answer of a kNN query. The rectangular region that bounds these cells transforms a kNN query into a range query.

After we determine the grid cells that contain the query answer, we determine a rectangular region that bounds these cells. Thus, we have transformed the kNN query into a range query, and hence our algorithms and techniques for splitting/merging and search can still handle kNN 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 9 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.

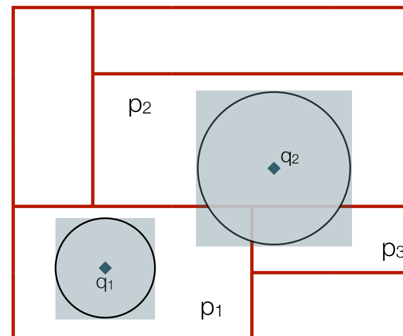


Figure 9: A kNN query is treated as a range query once the rectangular bounds enclosing the answer are determined.

Observe that the process of determining the region that encloses the k-nearest-neighbors of a query point is efficient. The reason is that the whole process is based on counting of main-memory aggregates without any need to scan any data points. Moreover, because the granularity of the grid is fine, the determined region is compact, and hence few partitions will be scanned during the execution of the kNN query, which leads to high query throughput.

6. SYSTEM INTEGRITY

6.1 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, a new query is received that may also trigger another split to the very same partitions being altered. Unless an appropriate concurrency control protocol is used, inconsistent partitioning will occur.

To address the above issue, we use a simple locking mechanism to coordinate the incremental updates of the partitions. In particular, whenever a query, say q, triggers a split, 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 split-queue. Because the split-queue resides in main-memory, updating the entries of queue is relatively 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 split-queue using a critical section.

6.2 Fault Tolerance

In case of system failures, the information in the main-memory structures might get lost which can affect the correctness of the query evaluation and the accuracy of the cost computations corresponding to Equation 1.

The main-memory grid contains two types of counts: 1) counts for the number of points, and 2) counts for the number of queries. When a new batch of data is received by AQWA, the counts of the number of points in the new batch are determined through a MapReduce job that automatically writes these counts into HDFS. Observe that the data points in the new batch are appended through the same MapReduce job. At this moment, the counts (of the number of points) in the grid are incremented and flushed into disk. Hence, the counts of the number of points are always accurate even if failures occur. Thus, the kNN queries are always answered correctly.

As mentioned in Section 3.2, the counts corresponding to the queries (i.e., rectangles) are periodically flushed to disk. Observe that: 1) the correctness of query evaluation does not depend on these counts, and 2) only the accuracy of the computation of the cost function is affected by these counts, which, in the worst case, leads to a delayed decision of repartitioning. In the event of failure before the counts are flushed into disk and if the query workload is consistently received at a certain spatial region, the counts of queries will be incremented and repartitioning will eventually occur.

7. EXPERIMENTS

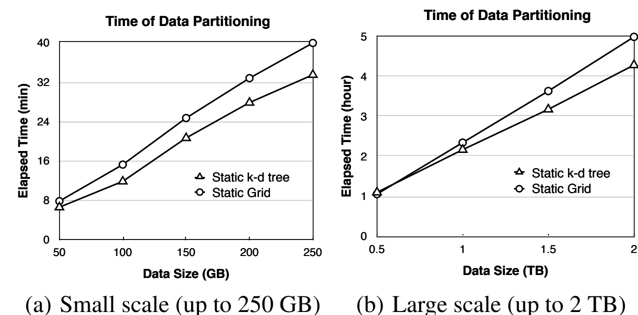
In this section, we evaluate the performance of AQWA. We realized a cluster-based testbed in which we implemented AQWA as well as static grid-based partitioning and static k-d tree partitioning (as in [19, 16]).[2] We choose the k-d and grid-based partitioning as our baselines because this allows us to contrast AQWA against two different extreme partitioning schemes: 1) pure spatial decomposition, i.e., when using a uniform grid, and 2) data decomposition, i.e., when using a k-d tree.

Experiments are conducted on a 7-node cluster running Hadoop 2.2 over Red Hat Enterprise Linux 6. Each node in the cluster is a Dell r720xd server that has 16 Intel E5-2650v2 cores, 64 GB of memory, 48 B of local storage, and a 40 Gigabit Ethernet interconnect. The number of cores in each node enables high parallelism across the whole cluster, i.e., we could easily run a MapReduce job with $7 \times 16 = 112$ Map/Reduce tasks.

We use a real spatial dataset from Twitter. The tweets were gathered over a period of nearly 20 months (from January 2013 to July 2014). Only the tweets that have spatial coordinates inside the United States were considered. The number of tweets in the dataset is 1.5 Billion tweets comprising about 250 GB. The format of each tweet is: tweet identifier, timestamp, longitude-latitude coordinates, and text. To better show the scalability of AQWA, we have further replicated this data 10 times, reaching a scale of about 2.5 Terabytes.

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. Because we are dealing with tweets in the United States, that has an area of 10 Million square kilometers, each grid cell in our virtual partitioning covers nearly 10 square kilometers, which is a fairly good splitting of the space given the large scale of the data. The virtual grid represents the search space for the partitions as well as the count statistics that we maintain.

7.1 Initialization



(a) Small scale (up to 250 GB) (b) Large scale (up to 2 TB)

Figure 10: Performance of the initialization process.

In this experiment, we study the performance of the initialization process of AQWA (Section 4.1). Figure 10 gives the execution time of the initialization phase for different sizes of the data. Observe that the initialization phase in AQWA is the same as that of a k-d tree partitioning. Hence, in the figure, we compare the performance of AQWA's initialization phase against the partitioning process of a uniform grid. For the grid-partitioning, we set the number of partitions to the number of partitions resulting from the k-d tree. Recall that in the initialization phase of AQWA, we apply recursive

[2] Our implementation is publicly available on GitHub [1].

k-d partitioning, and we stop when splitting a partition would result into small partitions, i.e., of size less than the block size in HDFS.

We observe that grid-partitioning requires relatively high execution time. Each reduce task handles the data of one partition. Due to the skewness of the data distribution, the load across the reduce tasks will be unbalanced, causing certain grid cells, i.e., partitions, to receive more data points than others. Because a MapReduce job does not terminate until the last reduce task completes, the unbalanced load leads to a relatively high execution time compared to the kd-tree. In contrast, the k-d tree partitioning, which is employed by AQWA, balances the data sizes across all the partitions.

We also observe that as the data size increases, the time required to perform the partitioning increases. As Figure 10(b) demonstrates, building the partitions from scratch for the whole data takes nearly five hours for only two Terabytes of data. Although this is a natural result, it motivates AQWA's incremental methodology in repartitioning, which is to avoid repartitioning the whole data throughout the system lifetime. In particular, after the initialization phase, AQWA never reconstructs the partitions again if new batches of data are received. In contrast, AQWA alters a minimal number of partitions according to the query workload and the data distribution.

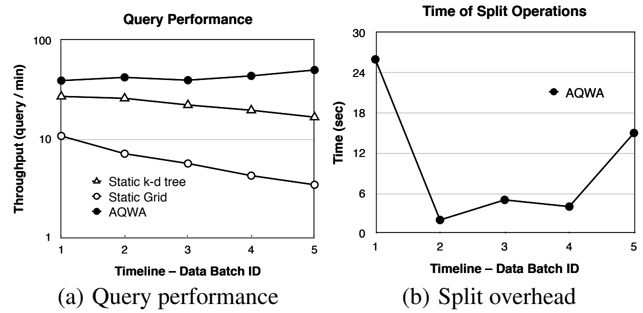
7.2 Adaptivity in AQWA

In the following experiments, we study the query performance in AQWA. Our performance measures are: 1) the system throughput, which indicates the number of queries that can be answered per unit time, and 2) the split overhead, which indicates the time required to perform the split operations. To ensure full system utilization, we issue batches of queries that are submitted to the system at once. The number of queries per batch is 20. The throughput is calculated by dividing 20 over the elapsed time to process the queries in a batch.

7.2.1 Data Acquisition and Incremental Repartitioning

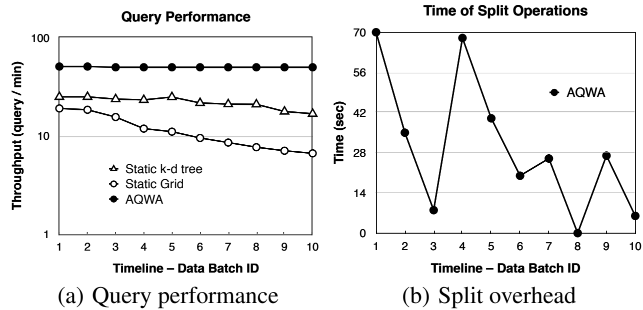
In this experiment, we study the query performance of AQWA after batches of data are appended through the data acquisition process. To simulate a workload with hotspots, we concentrate the queries over the areas that we identify as having relatively high data density. Figure 11 gives the query performance when batches of tweets are appended, where each batch is nearly 50 GB. Note that appending each batch of data takes almost the same time it takes to create partitions for 50 GB of data (i.e., first reading in Figure 10(a)). In this experiment, we focus on the query performance after the data is appended. Each point in Figure 11(a) represents the average performance for 50 batches of queries (each of 20 queries). Each point in Figure 11(b) represents the total time required for all the split operations for all the 50 batches, i.e., 1000 queries. Figure 12 repeats the same experiment, but for a higher scale, where 250 GB of data is appended at each batch.

As Figures 11 and 12 demonstrate, AQWA is an order of magnitude faster than a static grid-partitioning, and nearly 4 times faster than a static k-d partitioning. As more data is appended, the performance of both the static grid and k-d partitioning degrades because both are static. In contrast, AQWA maintains steady throughput regardless of the overall size of the data because it can dynamically split the partitions according to the query workload. One can argue that the static k-d partitioning could split as data is appended, however, this is quite costly because it requires reading and writing (almost) all the partitions (i.e., the entire data) from scratch. In contrast, AQWA determines a minimal set of partitions to split according to the query workload.



(a) Query performance (b) Split overhead

Figure 11: Performance with data updates for small scale data. Each data batch is 50 GB.

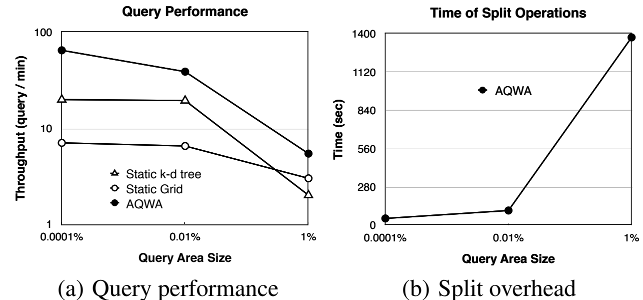


(a) Query performance (b) Split overhead

Figure 12: Performance with data updates for large scale data. Each data batch is 250 GB.

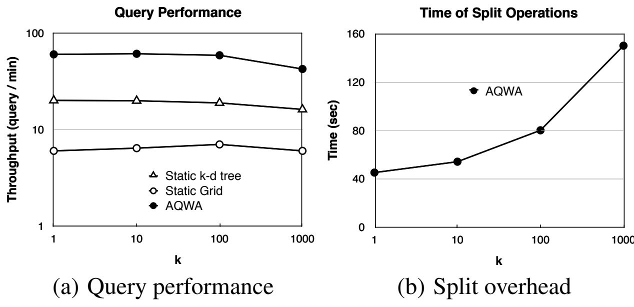
Figures 11(b) and 12(b) give the overhead incurred by AQWA in order to incrementally update the partitions. Notice that this overhead is in the scale of seconds, and hence is negligible when compared to the hours required to recreate the partitions from scratch (refer to Figures 10(b) and 10(a)). Furthermore, this overhead is amortized over 1000 queries. Observe that the split overhead is not consistently the same for each data batch because the distribution of the data can change from one batch to another, which directly affects the sizes of the partitions to be split.

In the above experiment, we have used range queries of size 5×5 according to our virtual grid, which is equivalent to 0.0025% of the area of the United States. Figure 13 gives the steady-state performance, i.e., at the last data batch, when varying the query region size. Observe that the performance degrades when the query area size increases. However, an area of 1% of the United States is fairly large (10,000 square kilometers). Figure 14 gives the performance of kNN queries for different values of k.



(a) Query performance (b) Split overhead

Figure 13: Performance of Range queries against the query region size.



(a) Query performance (b) Split overhead

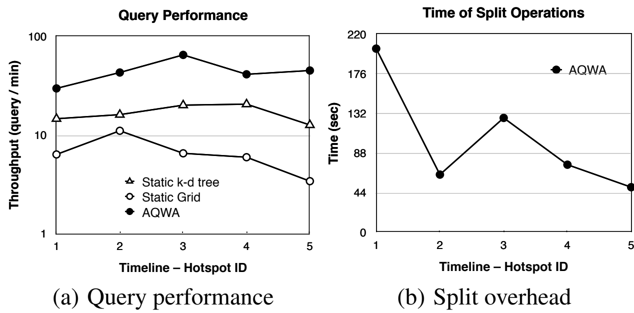
Figure 14: Performance of k-NN queries against the value of k.

7.2.2 Handling Multiple Query-Workloads

In this set of experiments, we study the effect of having two or more query-workloads. To have realistic workloads, i.e, where dense areas are likely to be queried with high frequency, we identify 5 hotspot areas that have the highest density of data. We have two modes of operation:

1. **Serial Execution:** In this mode, we simulate the migration of the workload from one hotspot to another. For instance, if we have 2000 queries and 2 hotspots, the first 1000 queries will be executed over one hotspot, followed by the other 1000 queries executed over the other hotspot.

2. **Interleaved Execution:** In this mode, queries are executed across the hotspots simultaneously, i.e., they are generated in a round-robin fashion across the hotspots. For instance, for 2000 queries and 2 hotspots, say h_1 and h_2, the first query is executed at h_1, the second query is executed at h_2, the third query is executed at h_1, and so forth.

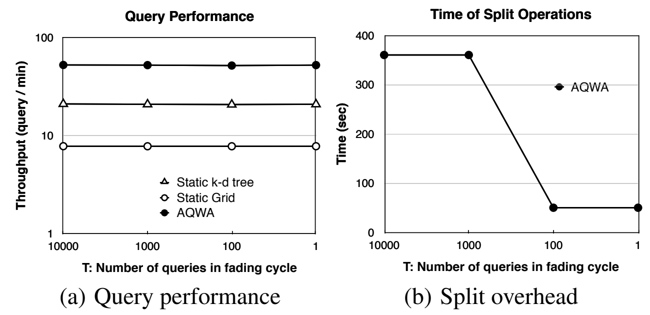


(a) Query performance (b) Split overhead

Figure 15: Serial execution of five different hotspots.

Figure 15 gives the performance for the serial execution of five hotspots, where each hotspot gets 1000 range queries of region 0.01% of the area of the United States. Observe that due to the skewness in the data, some hotspots may incur higher split overhead than others (Figure 15(b)). Similarly, the query throughput is not the same at all hotspots. However, for all hotspots, AQWA is superior to the static grid and k-d partitioning (Figure 15(a)).

As discussed in Section 4.2.4, AQWA differentiates between older workloads and the current workloads through the time-fading mechanism, which enables AQWA to avoid redundant split operations and reduce the overhead of repartitioning. To demonstrate the effect of varying the time-fading cycle T, we serially execute 2000 queries at two hotspots. After the queries in the first hotspot are executed, some queries are executed at the second hotspot, and at this



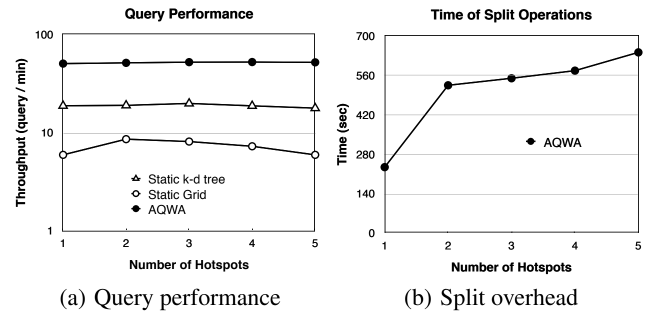
(a) Query performance (b) Split overhead

Figure 16: Performance against the time-fading cycle T for serial hotspot execution.

point a batch of data of 250 GB is received. Figure 16 gives the performance when varying the time-fading cycle T. As Figure 16(b) demonstrates, a small value of the fading-cycle (i.e., 100 queries or less) leads to a 5 times reduction in the split overhead. In contrast, a relatively large fading-cycle (i.e., 1000 queries or more) does not avoid that redundant overhead. In all cases, however, AQWA achieves superiority in terms of query throughput (Figure 16(a)) over both the static grid and k-d partitioning.

We have also experimented a mode of operation that is slightly different from the serial execution mode, where the transition from one hotspot to another is gradual. For instance, if we have two hotspots, say h_1 and h_2, some queries are executed first at h_1, then, afterwards, some queries are executed at h_1 and h_2 simultaneously. Finally, all the queries are executed at h_2 only. We find that this mode of operation yields (almost) the same performance as the serial mode. We omit the results due to space limitations.

Figure 17 gives the performance for the interleaved mode of execution of different numbers of simultaneous hotspots, where each hotspot receives 1000 range queries of region 0.01% of the area of the United States. As the number of simultaneous hotspots increases, the split overhead increases. However, the split overhead is amortized over thousands of queries. Moreover, the incremental repartitioning overhead is much smaller than the time required to recreate the partitions from scratch.



(a) Query performance (b) Split overhead

Figure 17: Performance against the number of hotspots.

8. CONCLUDING REMARKS

In this paper, we presented AQWA, an adaptive and query-workload-aware partitioning mechanism for large-scale spatial data. In AQWA we addressed several performance and system challenges; these include the limitations of Hadoop (i.e., the NameNode bottleneck), the overhead of rebuilding the partitions in HDFS, the dynamic nature of the data where new batches are created every day, and the issue of workload-awareness where not

only the query workload is skewed, but also it can change. We showed that AQWA successfully addresses these challenges and provides an efficient spatial-query processing framework. Using our implementation of AQWA on a Hadoop cluster, real spatial data from Twitter, and various workloads of range and kNN queries, we demonstrated that AQWA outperforms the state-of-the-art system by an order of magnitude in terms of query performance. Furthermore, we demonstrated that AQWA incurs little overhead (during the process of repartitioning the data) that is negligible when compared to the overhead of recreating the partitions.

Although our experimental evaluation is based on Hadoop, we believe that AQWA can be applied to other platforms. An example is Storm [2], a distributed platform for processing streaming data. In Storm, distributed processing is achieved using topologies of bolts (i.e., processing units), where the bolts can be connected according to a user-defined structure (not necessarily MapReduce). AQWA can dynamically reorganize the data in the bolts according to the query workload and the data distribution (e.g., see [29]). One of the limitations of Storm is that the number of bolts is fixed throughout the lifetime of a topology. Hence, in addition to the split operations, AQWA has to support merge operations in order to abide by that system limitation and keep a constant number of data partitions (i.e., bolts).

In our design of AQWA, we focused on spatial range and kNN queries, which cover a large spectrum of (useful) spatial queries. Extending AQWA to spatial-join queries is another potential future work. In particular, this demands extending AQWA's cost model to account for the overhead of communicating the data between the processing units (e.g., mappers and reducers) in order to compute a spatial join.

9. REFERENCES

[1] https://github.com/ahmed-m-aly/AQWA.git.

[2] Apache storm. https://storm.apache.org, 2015.

[3] Twitter statistics. http://www.internetlivestats.com/twitter-statistics/, 2015.

[4] D. Achakeev, B. Seeger, and P. Widmayer. Sort-based query-adaptive loading of r-trees. In *CIKM*, pages 2080–2084, 2012.

[5] 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.

[6] A. M. Aly, W. G. Aref, and M. Ouzzani. Cost estimation of spatial k-nearest-neighbor operators. In *EDBT*, pages 457–468, 2015.

[7] N. An, Z. Yang, and A. Sivasubramaniam. Selectivity estimation for spatial joins. In *ICDE*, pages 368–375, 2001.

[8] R. Beigel and E. Tanin. The geometry of browsing. In *LATIN*, pages 331–340, 1998.

[9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[10] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 3rd ed. edition, 2008.

[11] S. Chen. Cheetah: A high performance, custom data warehouse on top of MapReduce. *PVLDB*, 3(2):1459–1468, 2010.

[12] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[13] 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.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[15] G. Dröge and H. Schek. Query-adaptive data space partitioning using variable-size storage clusters. In *SSD*, pages 337–356, 1993.

[16] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in SpatialHadoop. *PVLDB*, 8(12):1602–1613, 2015.

[17] A. Eldawy and M. F. Mokbel. A demonstration of SpatialHadoop: An efficient MapReduce framework for spatial data. *PVLDB*, 6(12):1230–1233, 2013.

[18] A. Eldawy and M. F. Mokbel. Pigeon: A spatial MapReduce language. In *ICDE*, pages 1242–1245, 2014.

[19] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[20] 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.

[21] 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.

[22] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for MapReduce. *PVLDB*, 4(7):419–429, 2011.

[23] S. Geffner, D. Agrawal, A. El Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *ICDE*, pages 328–335, 1999.

[24] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[25] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, pages 73–88, 1997.

[26] L. Jiang, B. Li, and M. Song. The optimization of HDFS based on small files. In *IC-BNMT*, pages 912–915, 2010.

[27] H. Liao, J. Han, and J. Fang. Multi-dimensional index on Hadoop distributed file system. In *NAS*, pages 240–249, 2010.

[28] G. Mackey, S. Sehrish, and J. Wang. Improving metadata management for small files in HDFS. In *CLUSTER*, pages 1–4, 2009.

[29] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. Basalamah. Tornado: A distributed spatio-textual stream processing system. *PVLDB*, 8(12):2020–2031, 2015.

[30] 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.

[31] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.

[32] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[33] 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.

[34] S. Prabhakar, K. A. S. Abdel-Ghaffar, D. Agrawal, and A. El Abbadi. Cyclic allocation of two-dimensional data. In *ICDE*, pages 94–101, 1998.

[35] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *SPAA*, pages 78–87, 1998.

[36] M. Riedewald, D. Agrawal, and A. El Abbadi. Flexible data cubes for online aggregation. In *ICDT*, pages 159–173, 2001.

[37] M. Riedewald, D. Agrawal, A. El Abbadi, and R. Pajarola. Space-efficient data cubes for dynamic environments. In *DaWak*, pages 24–33, 2000.

[38] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[39] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.

[40] C. Sun, D. Agrawal, and A. El Abbadi. Selectivity estimation for spatial joins with geometric selections. In *EDBT*, pages 609–626, 2002.

[41] 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.

[42] K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-aware indexing of continuously moving objects. *PVLDB*, 2(1):1186–1197, 2009.

[43] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[44] S. Zhang, L. Miao, D. Zhang, and Y. Wang. A strategy to deal with mass small files in HDFS. In *IHMSC*, volume 1, pages 331–334, 2014.