A Demonstration of AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data[*]

Ahmed M. Aly[1], Ahmed S. Abdelhamid[1], Ahmed R. Mahmood[1], Walid G. Aref[1], Mohamed S. Hassan[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, samy, amahmoo, aref, msaberab}@cs.purdue.edu,
[2]hazem.elmeleegy@turn.com, [3]mouzzani@qf.org.qa

ABSTRACT

The ubiquity of location-aware devices, e.g., smartphones and GPS devices, has led to a plethora of location-based services in which huge amounts of geotagged information need to be efficiently processed by large-scale computing clusters. This demo presents AQWA, an adaptive and query-workload-aware data partitioning mechanism for processing large-scale spatial data. Unlike existing cluster-based systems, e.g., SpatialHadoop, that apply static partitioning of spatial data, AQWA has the ability to react to changes in the query-workload and data distribution. A key feature of AQWA is that it does not assume prior knowledge of the query-workload or data distribution. Instead, AQWA reacts to changes in both the data and the query-workload by incrementally updating the partitioning of the data. We demonstrate two prototypes of AQWA deployed over Hadoop and Spark. In both prototypes, we process spatial range and k-nearest-neighbor (kNN, for short) queries over large-scale spatial datasets, and we exploit the performance of AQWA under different query-workloads.

1. INTRODUCTION

Typical spatial query-workloads exhibit skewed access patterns, where certain spatial areas receive queries more frequently than others. As noted in several research efforts, e.g., [8, 4], accounting for the query-workload can achieve significant performance gains. Furthermore, the spatial distribution of the data can change over time. However, existing cluster-based systems for processing spatial data employ static partitioning schemes that are insensitive to the query-workload. For instance, SpatialHadoop [5] supports only static partitioning to handle spatial data in Hadoop. To handle a batch of data updates in SpatialHadoop, the entire data needs to be repartitioned from scratch, which is quite costly.

This demo presents AQWA, an adaptive and query-workload aware mechanism for partitioning large-scale spatial data. According to the query-workload and the data distribution, AQWA recursively divides the underlying space/data into partitions. In case of Hadoop, these partitions reside in disk as HDFS files. In case of Spark [11], these partitions reside in main-memory as Resilient Distributed Datasets [10] (RDDs, for short). Whenever a query is received, only the partitions (i.e., files in HDFS or RDDs in Spark) that are relevant to the query are processed. AQWA does not presume any knowledge of the query-workload or the data distribution. Instead, AQWA can detect, in an online fashion, the changes in the query-workload or data distribution, and accordingly reorganize the partitioning of the data.

Traditional spatial index structures try to increase the pruning power at query time by having (almost) unbounded decomposition until the finest granularity of the data is reached in each split. However, in a typical distributed system, e.g., HDFS, allowing too many small partitions can be very harmful to the overall health of the underlying cluster (e.g., see [2, 6, 7]). 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 that handles the file requests across the whole cluster. Similarly, in Spark, the metadata of the RDDs is shared in a centralized resource. Hence, 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.

Due to the scale of the data, the search space of the boundaries of the partitions is huge, and hence the process of finding the partitioning that would result in good performance gains is challenging. AQWA employs a cost function that models the cost of executing the queries. In addition, AQWA employs a set of main-memory structures that summarize the data distribution as well as the query-workload. These structures enable AQWA to efficiently perform its adaptive repartitioning decisions by supporting $O(1)$ computation of the cost function.

We will demonstrate AQWA based on two prototypes deployed on top of Hadoop and Spark using large-scale datasets from Twitter, and different query-workloads of spatial range and kNN queries.

2. OVERVIEW OF AQWA

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. 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:

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

[*]This research was supported in part by National Science Foundation under Grants IIS 0916614, IIS 1117766, and IIS 0964639.

Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

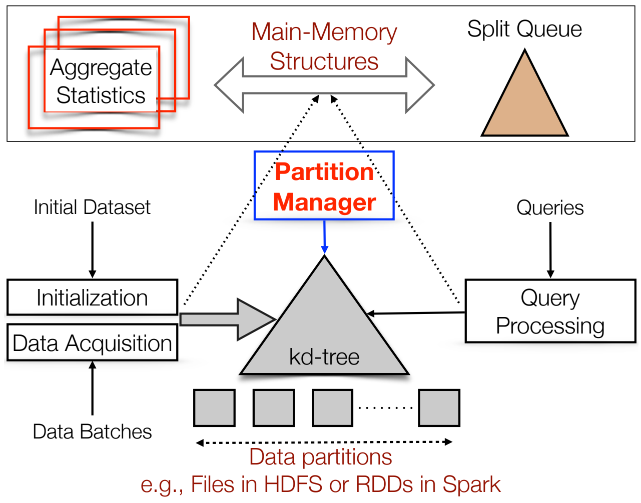


Figure 1: An overview of AQWA.

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 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, and 2) a set of main-memory structures that maintain statistics about the distribution of the data and the queries. Four main processes define the interaction between the components of AQWA, namely, Initialization, Query Execution, Data Acquisition, and Repartitioning.

1. **Initialization:** During this process, we collect statistics (i.e., counts) about the data distribution, and accordingly, create an initial partitioning of the data.

2. **Query Execution:** This process selects the partitions that are relevant to, i.e., overlap, the invoked query. The answer of the query is determined from the selected partitions.

3. **Data Acquisition:** Given a batch of data, we issue a MapReduce job (or a Spark job) that appends each new data point to its corresponding partition according to the current layout of the partitions. In addition, the counts collected in the Initialization process are incremented to the given batch of data.

4. **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.

Below, we list the performance challenges that the above four processes raise, and briefly describe how they are addressed in AQWA.

- *Overhead of rewriting:* After a batch of data is appended, some partitions may need to be split in order to have good pruning power at query time. If the process of altering the partitioning reconstructs the partitions from scratch, it would be very inefficient because it will have to read and write the entire data. Hence, AQWA adopts an incremental mechanism that alters only a minimal set of partitions according to the query-workload.

[1]AQWA is applicable to R-Tree or quadtree decomposition, but the demo is based on a k-d tree implementation.

- *Efficient search:* AQWA repeatedly searches for the partitions to be split in order to achieve good query performance according to Equation 1. 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. As we explain in Section 3, AQWA maintains a set of main-memory aggregates to efficiently determine the best splits via $O(1)$ main-memory lookups.

- *Keeping a lower bound on the size of each partition:* Allowing too many small partitions can introduce a performance bottleneck in a distributed file system (e.g., see [2, 6, 7]). Hence, AQWA avoids splitting a partition if either of the two resulting partitions is of size less than the block size in the distributed file system (e.g., 128 MB in HDFS).

- *Time differentiation between queries:* Although AQWA keeps the history of all the queries that have been processed, AQWA differentiates between fresh queries (i.e., those that belong to the current query-workload) and relatively old queries. This is achieved through a time-fading mechanism that assigns lower weights in the cost corresponding to older queries. In turn, this alleviates the redundant repartitioning overhead corresponding to older query-workloads.

- *Concurrency control:* As queries are received by AQWA, some partitions may need to be split. It is possible that while a partition is being split, a new query is received that may also trigger another change to the very same partition being split. Unless an appropriate concurrency control protocol is used, inconsistent partitioning can occur. To solve this problem, we use a simple locking mechanism that coordinates the incremental updates of the partitions.

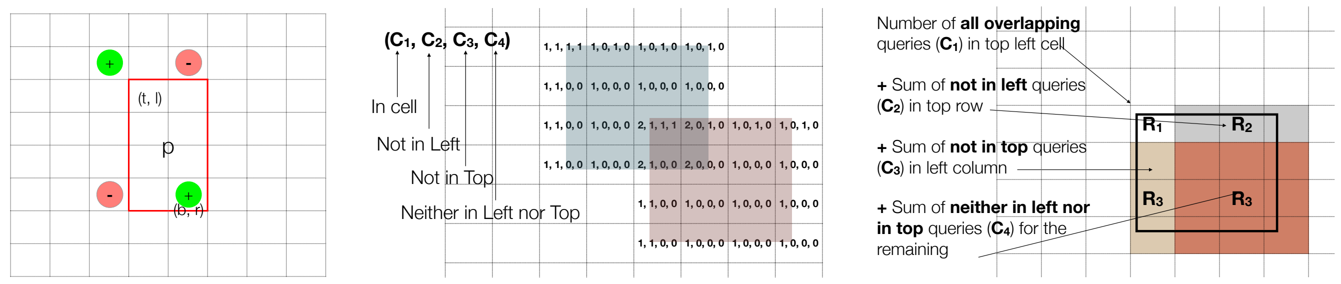
3. ADAPTIVITY IN AQWA

3.1 Data Acquisition

AQWA is most useful in dynamic scenarios where new batches of data are received frequently. For instance, consider the Twitter dataset, where 500 Million tweets are created everyday [1]. During the Data Acquisition process, each of the partitions is appended with a new set of data points during. In case of Hadoop, this is performed through a MapReduce job, where in the Map phase, the partition of each data point is determined, and in the Reduce phase, the data points are appended to the corresponding partitions. In case of Spark, we use a Spark job to achieve the same behavior. We use two transformation: the first transformations is to find the partition of each data point, and the second transformation is to merge the data points with the corresponding partitions.

After the data is appended, some (if not all) partitions may increase in size. In order to have good pruning at query time, these partitions need to be split. A straightforward approach 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 scale of the data. Furthermore, this approach may require the entire system to halt until the new partitions are created.

To solve this problem, AQWA applies 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. A partition, say p, is split into two partitions, say p_1 and p_2, if the performance



(a) Finding the number of points in a rectangle in $O(1)$.

(b) The counter values corresponding to two range queries.

(c) Finding the number of queries overlapping a rectangle in $O(1)$.

Figure 2: An illustration of the main-memory aggregates maintained for the data and query-workload distributions.

gain (according to Equation 1) resulting from splitting p is expected to be greater than the split overhead, i.e., reading p and writing p_1 and p_2.

AQWA repeatedly tries to find the best horizontal/vertical lines at which each partition needs to be split, and prioritizes all the partitions in a priority queue according to the expected gain resulting from further splitting. Hence, two operations need to be efficiently supported, namely, finding the number of points in a partition, and finding the number of queries that overlap a partition. Below, we show efficient techniques that support these two operations in $O(1)$.

We divide the space into a fine-grained grid, say G, of n rows and m columns. Each grid cell, say $G[i,\ j]$, maintains aggregate statistics about the data distribution as well as the query-workload. The grid is kept in main-memory and is chosen to have very fine granularity.

3.2 Statistics for Data Distribution

During the Initialization process, we count the number of points in each grid cell. This is achieved through a MapReduce job (or Spark job) that reads the entire data and determines the count of points in each grid cell. Afterwards, we aggregate the counts in each grid cell as follows. For every row in G, we scan the cells from column 0 to column m and aggregate the values as we go, i.e., $G[i,\ j] = G[i,\ j] + G[i,\ j-1]\ \forall\ j \in [2,\ m]$. Afterwards, we repeat the same process on the column level, i.e., $G[i,\ j] = G[i,\ j] + G[i-1,\ j]\ \forall\ i \in [2,\ n]$. At this moment, the value at each cell, say $G[i,\ j]$, corresponds to the number of points in the rectangle bounded by $G[0,\ 0]$ (top-left) and $G[i,\ j]$ (bottom-right).

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 2(a) illustrates, the number of points, say N_p, in Rectangle(b, r, t, l) is $N_p(b,\ r,\ t,\ l) = G[b,\ r] - G[t-1,\ r] - G[b,\ l-1] + G[t-1,\ l-1]$. Observe that this $O(1)$ lookup operation is key for the efficiency of AQWA during the Initialization as well as the Repartitioning processes. Without the main-memory aggregate counts, other alternatives, e.g., successively scanning the data in order to find the number of points in a rectangle, would be impractical due to the large scale of the data.

Whenever data updates are received, we process these updates in batches, and update the counts in the grid accordingly.

3.3 Statistics for Query-Workload

At each grid cell, say $G[i,\ j]$ we maintain four additional counters. Refer to Figure 2(b) for illustration.

- 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).

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, 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, 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 2(a)). 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 fine-grained grid that maintains the counts resides in main-memory, and hence the fine-grained grid is cheap to update.

A partition, say p, can be divided into four regions R_1 through R_4 as illustrated in Figure 2(c). 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.

The sum of the above values represents the number of queries that overlap p.

3.4 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 within the query itself. In

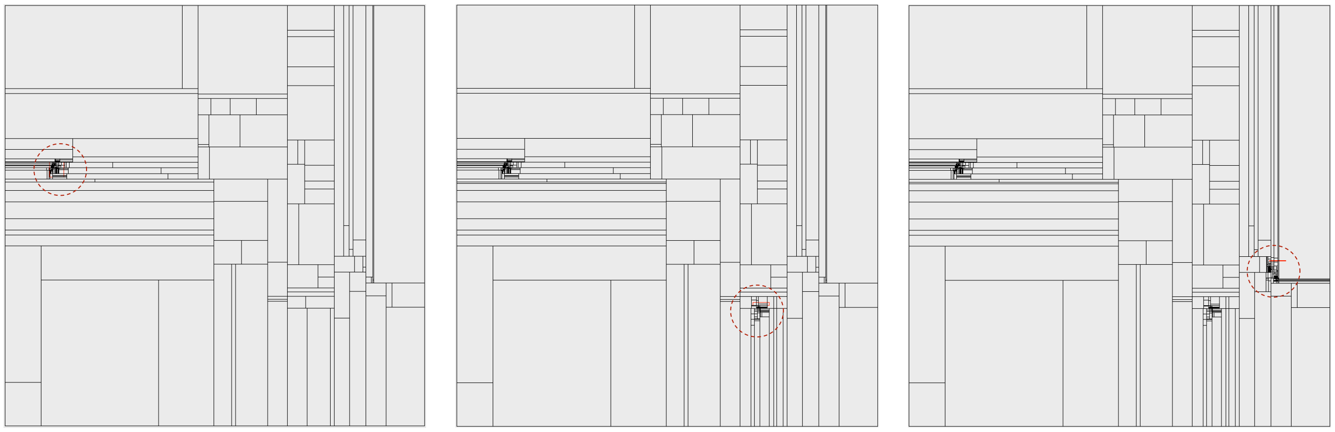

(a) Partitioning at t_0. (b) Partitioning at t_1. (c) Partitioning at t_2.

Figure 3: Interactive partition visualizer. A display of the layout of the partitions when the query-workload moves over time from one hotspot to another. The partitions are aggressively split at the region of the hotspot.

contrast, the boundaries that contain the answer of a kNN query are unknown until the query is executed. 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 [3])

To address this problem, we make use of the fine-grained grid that contains statistics about the data distribution and query-workload. Given a k-NN query, we determine the grid cells that are guaranteed to contain the answer of the query using the MINDIST and MAXDIST metrics as in [9]. In particular, we scan the 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. The rectangular region that bounds the scanned cells maps the kNN query into a range query.

Observe that the process of determining the region that encloses the kNN is efficient because it is based on counting of main-memory aggregates, and it does not require scanning any data points. Moreover, because the granularity of the grid is fine (compared to that of the partitions), the determined region is compact.

4. DEMO SCENARIO

We will demonstrate AQWA and showcase its performance using two prototypes deployed over Hadoop and Spark. We use a real spatial dataset from Twitter. The dataset was gathered over a period of nearly two years, and only the tweets that have spatial coordinates inside the United States were considered. We will show how AQWA can consume batches of tweets, and yet, maintain steady query performance using its adaptive repartitioning mechanism.

As part of our prototypes, we implement an interactive visualizer that displays the layout of the partitions over time. Figure 3 displays the layout of partitions under a certain query-workload that moves across the map as illustrated in Figures 3(a), 3(b), and 3(c), respectively. It is clear from the figure how AQWA adaptively updates the partitioning of the data according to the changes in the query-workload.

We will show the performance of AQWA under different query-workloads of spatial range and kNN queries. We will use different setups for the query-workloads, where the workload can shift from one spatial hotspot to another, and when there are multiple simultaneous hotspots. We will compare the performance of AQWA against static partitioning structures, and show how AQWA can achieve orders of magnitude gain in query performance.

5. REFERENCES

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

[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*, pages 457–468, 2015.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.