SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data

Ahmed Eldawy[1], Mohamed F. Mokbel[1], Saif Alharthi[2], Abdulhadi Alzaidy[2], Kareem Tarek[2], Sohaib Ghani[2]

[1]*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA*
{eldawy, mokbel}@cs.umn.edu
[2]*KACST GIS Technology Innovation Center, Umm Al-Qura University, Saudi Arabia*
{sharthi, azaidy, ktarek, sghani}@gistic.org

***Abstract*—Remote sensing data collected by satellites are now made publicly available by several space agencies. This data is very useful for scientists pursuing research in several applications including climate change, desertification, and land use change. The benefit of this data comes from its richness as it provides an archived history for over 15 years of satellite observations for natural phenomena such as temperature and vegetation. Unfortunately, the use of such data is very limited due to the huge size of archives ($> 500TB$) and the limited capabilities of traditional applications. This paper introduces SHAHED; a MapReduce-based system for querying, visualizing, and mining large scale satellite data. SHAHED considers both the spatial and temporal aspects of the data to provide efficient query processing at large scale. The core of SHAHED is composed of four main components. The *uncertainty* component recovers missing data in the input which comes from cloud coverage and satellite mis-alignment. The *indexing* component provides a novel multi-resolution quad-tree-based spatio-temporal index structure, which indexes satellite data efficiently with minimal space overhead. The *querying* component answers *selection* and *aggregate* queries in real-time using the constructed index. Finally, the *visualization* component uses MapReduce programs to generate heat map images and videos for user queries. A set of experiments running on a live system deployed on a cluster of machines show the efficiency of the proposed design. All the features supported by SHAHED are made accessible through an easy to use web interface that hides the complexity of the system and provides a nice user experience.**

I. INTRODUCTION

Several space agencies such as National Aeronautics and Space Administration (NASA) [1] and European Space Agency (ESA) [2] collect enormous amounts of remote sensing data using satellites that continuously orbit the earth. For example, the Land Process Distributed Active Archive Center (LP DAAC) provided by NASA contains more than 500TB of data and is increasing on a daily basis [3]. This archive contains historical satellite data of a dozen of natural phenomena including temperature, vegetation, surface reflectance, and thermal anomalies, for every 250 square meter[2] for the whole world over the last 15 years. The archive is updated on a daily basis with newly collected data. Such archive is very useful and is actually being used by meteorologists and other scientists in several important applications, including detection of desertification [4], studies of land cover change [5], understanding ocean dynamics [6], and more generally in climate informatics [7] as well as in various governmental initiatives about climate change, e.g., see [8]–[10].

Although they are very useful, the huge size of such archives makes it hard for scientists and researchers to use. As a result, all prior research analysis attempts are either done offline (i.e., query response time may take hours or days) or based on a very small sample of the whole archive. A standard way to use the LP DAAC archive is to go through a web interface (e.g., Reverb [11]) in which the user provides some selection criteria including spatio-temporal predicates. Then, a list of files that match this selection criteria is returned. Such web interface methods are not efficient for many queries. For example, a simple selection query such as "What is the daily temperature of a specific location in the year of 2013?" would select 365 files which contain around 500 million points with a total download size of 2GB. All this data will need to be processed to select only 365 points that provide the query answer. What is even more challenging here is that the data returned from these files may be missing important information either because of the satellites misalignment or because the reading area was covered by clouds at the time the image is taken. Hence to answer such query, data has to be cleaned first on-the-fly before responding to the query. Such challenges and overhead make it very hard to use such available rich satellite data.

In this paper, we present SHAHED; a system that lays out the necessary infrastructure to query, mine, and visualize big spatio-temporal satellite data. In particular, SHAHED downloads its data on a daily basis from the LP DAAC archive, resolves its data uncertainty, and locally indexes the downloaded data, making it ready for querying, mining, and visualization. SHAHED had to face two contradicting challenges; the need to process large-scale satellite data (in order of tens of tera bytes) and the need to provide real-

[1]This work was done while these two authors were visiting the GIS Technology Innovation Center in Umm AlQura University and is supported by the center under project GISTIC-13-05

[2]Some datasets have a lower resolution with data for every one KM^2

time query response time. Large-scale data processing calls for relying on a MapReduce-based environment to allow an elastic computing environment that employs a large number of computational nodes. Meanwhile, real-time query response calls for *not* using such MapReduce environments due to their overhead in initiating job requests. Generally, MapReduce environments are made for batch queries rather than online queries. As a result, SHAHED deploys SpatialHadoop [12], a MapReduce framework for spatial data, for all of its offline functionality, which is needed to build rich and powerful index structures. Then, a separate query engine is used to retrieve the answer from the already built index structures without going through any MapReduce environment.

SHAHED is divided into two main sets of components; *data interface* and *user interface* components. The *data interface* is basically a background process that wakes up once everyday (e.g., at midnight) to download newly added data from NASA LP DAAC archive. Then, it triggers the execution of the following two consecutive modules: (1) The *uncertainty handling* module, which goes through the downloaded data to fill in the missing information. This is done by developing a two-dimensional smoothing technique over missing information. For efficient uncertainty processing, we use SpatialHadoop to scale up this module. (2) The *indexing* module, which takes the cleaned data from the *uncertainty handling* module, builds a spatial index for this data (using SpatialHadoop), and appends this new index to the current list of available spatial index structures in a way that forms a global spatio-temporal index over all available satellite data. In addition to being triggered daily, the *indexing* module is also triggered monthly and yearly, to combine the set of daily and monthly indexes into one bigger spatial index structure that covers data for a whole month and year, respectively. This is still done within the main spatio-temporal index structures maintained by SHAHED *indexing* module.

The *user interface* of SHAHED receives three kinds of requests from its users, namely, querying, mining, and visualization requests. Each request goes to a corresponding module. Hence, the user interface of SHAHED is composed of three main module, *querying*, *mining*, and *visualization*. In this paper, we focus and discuss only the *querying* and *visualization* modules, while the *mining* module is out of scope of this paper. The *querying* module supports two kinds of queries: (1) spatio-temporal *selection* queries, where users can request a set of values (e.g., temperature or vegetation) for a certain spatial region over a certain temporal interval, and (2) spatio-temporal *aggregate* queries, which is similar to selection queries, yet we report the aggregate (e.g., average or maximum) of all the values within the specified spatio-temporal range. For higher efficiency, the *querying* module does not go through SpatialHadoop. Instead, it has its own separate query engine that returns the query answer efficiently in an interactive real-time response time; something that cannot be provided should we go with a MapReduce environment. This is achieved by exploiting the spatio-temporal index structure, built and maintained by the *indexing* module.

The *visualization* module supports two main functionality: (1) *spatio-temporal heat maps*, where users can request to generate a sequence of heat maps of certain values (e.g., temperature or vegetation) for a certain spatial region and over a certain temporal period. The sequence of heat maps are returned as a set of images as well as an animated video. This is a very important and needed functionality by meteorologists as they need to visualize the change of behavior over a certain temporal period, (2) *multi-level spatial heat maps*, where users can request a single multi-level heat map for a certain time instance over a certain spatial region. Such multi-level heat map allows the user to zoom in and out within the picture to get either lower or higher resolution heat maps in an interactive way, which is another important functionality requested by meteorologists. Unlike the *querying* module, the *visualization* module is not interactive. Instead, it works as a web service, where users submit their requests through a nicely designed web interface. Once a request is submitted, SHAHED exploits its index structure to retrieve the required data while generating the requested heat maps. Once this process is done, an email is sent to the user as a notification that the request is finished with a link to download the requested data. The time to satisfy a request heavily depends on the size of the area covered by the request and the length of the temporal period. Requests with large areas (e.g., the whole world) over a long time period (e.g., a whole year) may take an hour or so to satisfy. That is still acceptable as usually such requests do not need real-time response. This also allows us to comfortably use SpatialHadoop to scale up the heat map generation.

Our reported experience and use of SHAHED show that it is a very efficient system with a wide use. In terms of execution time, the *uncertainty* and *indexing* modules run as background processes triggered periodically. The process usually takes up to few minutes, and it does not affect the query performance. In the mean time, spatio-temporal selection and aggregate queries within the *querying* module are all supported with a real-time response time. This is mainly due to the fact that they have their own path to exploit the already built spatio-temporal index structure. Finally, generating a heat map within the *visualization* module may take up to few minutes. For example, we have generated a single heat map for the whole world (using 450 Million points) in less than three minutes using a cluster of only four quad-core machines. Such numbers show the scalability, efficiency, and usability of SHAHED system.

The rest of this paper is organized as follows. Section II gives a background overview of NASA satellite data. The system overview of SHAHED is presented in Section III. The *uncertainty*, *indexing*, *querying*, and *visualization* modules are described in Sections IV, V, VI, and VII, respectively. In Section VIII, we report experimental numbers from our use of SHAHED. The web interface of SHAHED is highlighted in Section IX. Finally, Section X concludes the paper.

II. Background

The Land Process Distributed Active Archive Center (LP DAAC) [3] stores historical satellite data of a dozen of natural phenomena including temperature, vegetation, surface reflectance, and thermal anomalies. This section gives a necessary background of such data archive.

A. Structure and Format of the LP DAAC Archive

The LP DAAC archive is organized in a hierarchical structure that makes it easy to locate files by dataset, time, and location. Figure 1 illustrates the structure of files in the LP DAAC archive organized in four levels. In the first level, files are organized by their data sets, where each data set is stored in a separate directory (e.g., temperature or vegetation). In the second level, each data set is temporally partitioned in daily partitions, each stored in a separate directory named by the day of this snapshot. In the third level, data in each snapshot is partitioned using a uniform grid over the whole globe. Each grid tile is identified by its two-dimensional coordinate in the grid, e.g., h21v06 represents the cell in column 21 and row 6. In the fourth level, each file contains a two-dimensional array of numbers, which represent the values (e.g., temperatures) for each point in the given region and time. Files are of the Hierarchical Data Format (HDF), which is a binary format where readings are arranged in a two-dimensional array that covers the associated tile. The size of the array is either 1200×1200, 2400×2400, or 4800×4800 depending on the resolution of the data set, where each value in the array represents an area of size $1000 \times 1000m$, $500 \times 500m$, and $250 \times 250m$, respectively.

The coordinates of each value in the array are not explicitly stored, but it can be computed using the sinusoidal projection as follows: Given the temperature data set where each tile is of size 1200×1200 and a point in tile h21v06 at the position (100, 100) in the two-dimensional array. To compute its latitude and longitude coordinates, we first calculate the point location in the sinusoidal space as: $x = 21 + 100/1200$ and $y = 6 + 100/1200$. Then, the latitude and longitude are computed as: $lat = (9 - y) \times 10$ and $lon = (x - 18) \times 10 \times \cos(lat)$. The same equations can be reversed to compute the position of a point in a file given a latitude and longitude offsets.

B. Data Retrieval from LP DAAC Archive

With its current hierarchical organization, LP DAAC provides a simple way to retrieve a certain value given the type of the data set, a temporal range, and a spatial range. First, the list of directories is scanned to locate the directory of the requested dataset type. Then, the directories are kept sorted by time and the given time range is translated to a range of directories to access. Finally, in the third level of the hierarchy, a two-dimensional grid index is constructed *on the fly*, which makes it easy and efficient to select the files that match the user specified spatial range. To build the spatial grid index, each tile has to be assigned a spatial range according to the tile identifier (e.g., h21v06). The spatial range is calculated using the sinusoidal projection, described in Section II-A.

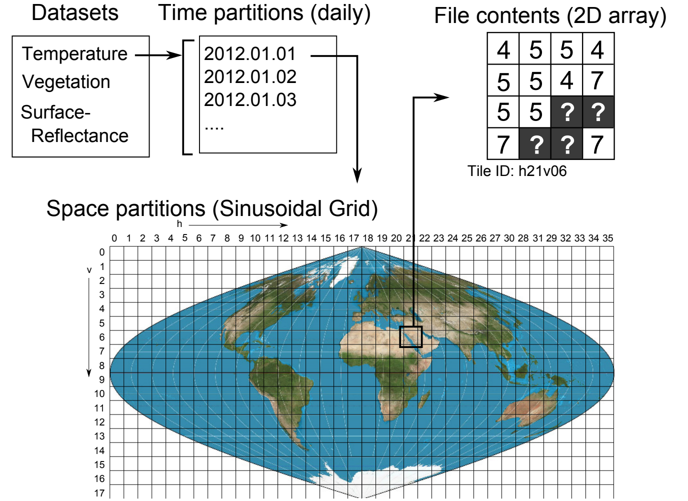


Fig. 1. Structure of LP DAAC archive

Unfortunately, such straightforward method is extremely inefficient when selecting a large set of data as this requires reading large number of files as described in an example in Section I. It is also inefficient when reading a single value as an on-the-fly index will be built while a large file is retrieved just to get a single value. Such inefficiency makes it hard for scientists to access such valuable archive. This becomes our main motivation to develop SHAHED to make such valuable archive easily accessible to scientists.

C. Data Uncertainty in LP DAAC Archive

A main challenge in processing NASA LP DAAC files is the data uncertainty imposed by *missing* values. Missing values are mainly a result of one of the following four reasons: (1) Data in regions outside the earth. Depending on the angle in which the image was taken by the satellite, part of this image might be outside earth. This type of missing values can be detected when a point is converted from sinusoidal space to latitude-longitude space as it produces an invalid longitude value, i.e., less than -180 or larger than 180. (2) The specified data set is available only for land (e.g., land temperature) while the missing point is located in a water area (e.g., ocean). This type of missing data is detected by imposing a water mask and detecting points that fall within water areas. The water mask is provided by NASA as a set of HDF files at the highest available resolution, i.e., 250×250 meters. (3) Mis-alignment of satellites results in uncovered sharp strip areas of the earth. The strips may cover different areas of the globe based on earth and satellite movements. (4) The satellites were not able to read values of certain areas as it was covered by the clouds at the time of taking the snapshot. All missing values of the four above types are marked in the LP DAAC files by a special (fill) value. These missing values should be handled and cleaned to avoid any incorrect computations. In SHAHED, we do so by skipping the values of the first two types as they are truly irrelevant and should be missing, while we clean the data from the last two types using SHAHED *uncertainty* module.

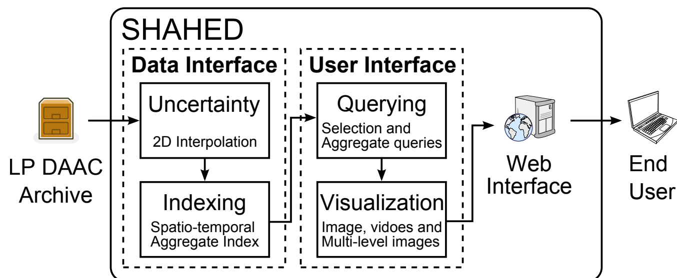
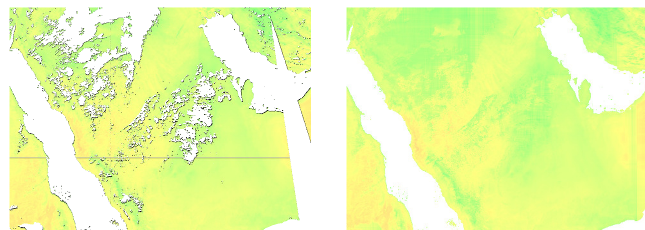


Fig. 2. Overview of SHAHED



(a) With missing data (b) Missing data recovered

Fig. 3. An example of hole recovery with a heat map

III. Overview

Figure 2 gives an overview of SHAHED system architecture, which consists of four main modules, namely, *uncertainty*, *indexing*, *querying*, and *visualization*, described below:
The uncertainty module. This module is triggered on a daily basis to clean the newly downloaded daily data from NASA LP DAAC archive. The objective is to estimate and recover the missing values of the satellite data that emerge either from satellite mis-alignment or cloud coverage. To do so, the *uncertainty* module employs a two-dimensional interpolation function that smooths out the missing values within the two-dimensional space. More details are described in Section IV.
The indexing module. This module employs a novel multi-resolution spatio-temporal index for efficient data indexing and retrieval. It is triggered by two events: (1) The end of the *uncertainty* module, where the new cleaned data is added to the current spatio-temporal index structure, and (2) Periodic monthly and yearly execution to compact the index structure. More details are described in Section V.
The querying module. This module receives spatio-temporal selection and aggregate queries from SHAHED users. Then, it exploits the spatio-temporal index structure with early pruning techniques to return the requested answer. More details are described in Section VI.
The visualization module. This module runs on top of the *querying* module to generate snapshot, multi-resolution, or animated heat maps based on a user query request. It employs novel visualization techniques that scale up the image generation process using the underlying MapReduce environment. More details are described in Section VII.

SHAHED also has one more module, namely, *mining* module which is *not* depicted in Figure 2, though it should lie between the *querying* and *visualization* modules. The *mining* module supports more complex and analysis queries, e.g., “find any outliers in a specific area over a certain range of time”, or “Given a set of dates and areas of past earthquakes, find out if there is a certain pattern that appears before earthquakes”. The *mining* module is out of scope of this paper, and hence we are not discussing it further, as it requires too much space to fill in the details of mining algorithms.

SHAHED is equipped with an easy-to-use map-based *web interface* layer that hides the complexity of the system through a simple and elegant web interface that accesses all SHAHED functionality. Details of the web interface are described in Section IX.

IV. Uncertainty

This section describes the *uncertainty* module in SHAHED. We start by showing the effect of uncertainty on satellite data, then, we present our algorithm to recover such uncertain data.

A. Data Uncertainty

Figure 3 depicts the plotting of two heat maps for the same area of Saudi Arabia on the same day. The first plotting (Figure 3(a)) relies on the raw LP DAAC data that includes a lot of missing values, while the second plotting (Figure 3(b)) relies on the same set of data, yet, after applying our uncertainty recovering technique. Focusing on the figure with uncertain data (Figure 3(a)), we can easily distinguish the effect of two different factors: (1) blank curvy polygon areas in the middle and top left of the figure, which is a result of areas covered by clouds at the time of taking the satellite snapshot, and (2) a blank sharp rectangle coming from the bottom right corner of the figure and going close to the top of the figure, which is a result of satellite misalignment.

B. Recovering Uncertain Data

Per Figure 3(a), it is clear that having blank areas and missing information in satellite data significantly reduces its usage. As a result, we have developed a simple data recovery technique that aims to predict the missing values using a two-dimensional interpolation function. The basic idea is to calculate two estimates for each missing point, namely, x-estimate and y-estimate, which are calculated using a traditional linear interpolation function based on the nearest two valid points on the same horizontal and vertical lines, respectively, as the missing point. Then the estimated value is computed by taking the average of the two estimates.

Figure 4 gives an example of how the two-dimensional interpolation technique works. All cells marked with a question mark or x represent a missing value. Empty cells are non-relevant to this example and are omitted for clarity. The missing value x_1 is estimated by taking the average of the x-estimate $= \frac{5\times3+9\times1}{4} = 6$ and the y-estimate $= \frac{4\times2+7\times1}{3} = 5$. The x-estimate is computed using a traditional interpolation function of the two values 5 and 9 with distances 1 and 3, respectively. Similarly, the y-estimate is calculated from the values 4 and 7 with distances 1 and 2, respectively. For x_2, there is no valid value on the same row left to it. Thus, we compute the x-estimate as 7, as the nearest value on the same row, which is then averaged with the y-estimate as before.

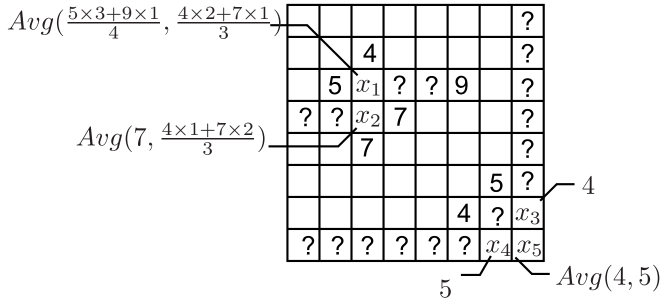


Fig. 4. Two-dimensional interpolation

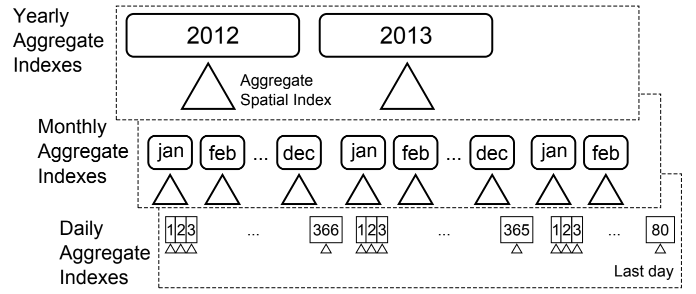


Fig. 5. Spatio-temporal aggregate index

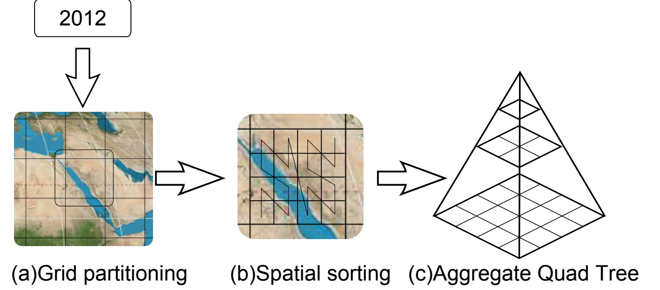


Fig. 6. Aggregate spatial index

x_3 does not have a y-estimate as there are no other values on the same column which means it is estimated using only the x-estimate. Similarly, x_4 is estimated using only the y-estimate as there are no other values on the same row. Finally, for x_5, there are no valid values either on the same row or column. For this special case, we compute its estimate after x_3 and x_4 are estimated by taking the average of x_3 and x_4. Figure 3(b) depicts the heat map after filling all missing values using our recovery technique. This makes querying, mining, and visualizing satellite date much beneficial.

V. Spatio-temporal Indexing

As discussed in Section II, the only available way to access LP DAAC data is through using on-the-fly indexes. However, this may end up to be very inefficient, even for simple queries. For example, retrieving all temperature values in a specific point over a period of 100 days would retrieve 100 files just to read one value from each file, which is extremely inefficient. Furthermore, aggregate queries such as computing the average temperature over a given area will retrieve a lot of points in this area before computing their average. To overcome such inefficiency, we equip SHAHED with a spatio-temporal index structure that is designed mainly to support the main SHAHED functionality of supporting spatio-temporal selection and aggregate queries as well as the visualization functionality.

Figure 5 gives the layout of our spatio-temporal index structure. The index has two orthogonal hierarchies as follows: **Temporal hierarchy**. The index is organized in three temporal layers, each representing the whole dataset using a different temporal resolution. The lowest resolution layer contains yearly index structures, i.e., the whole data of one year is included in one index, while the highest resolution layer contains daily index structures. A monthly data structure is built only after the whole month is concluded. Similarly, the yearly index structure is built only at the end of the year. Hence, in Figure 5, we can see that there are 80 daily index structures from 2014, yet, only two monthly index structures for January and February, as the March index is not built yet. Similarly, the 2014 index is not built yet. It is important to note that indexes in one level are independent of indexes in other levels, which means that data is replicated three times. This replication is the storage overhead we choose to pay to provide efficient query processing, as will be described later.

Spatial hierarchy. Figure 6 gives the details of each yearly, monthly, or daily index structure. Each index by itself is an aggregate spatial index in the form of an aggregate quad-tree [13]. The lowest level of the aggregate spatial index partitions the data spatially using a uniform grid (Figure 6(a)). The reason we use a grid partitioning is that the data in NASA LP DAAC archive is uniformly distributed. Then, in each partition, the points are sorted using their respective Z-order values within the partition (Figure 6(b)). Finally, on top of the sorted points, we build an aggregate quad tree [13], which can be efficiently constructed when the points are already sorted [14]. The quad tree is prepended to the data file to ensure they are both stored in the same machine in HDFS. Each node in the aggregate quad tree includes a list of aggregates, namely, sum, minimum, maximum, and count, of all entries in its children nodes. Other aggregate values can be derived from the cached values such as average ($sum/count$) or range ($maximum - minimum$), and more functions can be added such as *variance* or *standard deviation*.

When the system is deployed for the first time, the index is bulk loaded with all data that is already available in the LP DAAC archive. This bulk loading process starts by building all the indexes in the daily temporal layer. Then, it builds indexes for the larger time interval (i.e., monthly) by merging daily indexes for all previous (i.e., completed) months. After that, it builds yearly spatial indexes for all past years by merging monthly spatial indexes. Once the index is bulk loaded with all existing data, the indexing module is then called at regular time intervals to add new daily snapshots of data as they are added to the LP DAAC archive.

To be able to realize such indexing layout, there are three

main components that are used as building blocks to realize the spatio-temporal indexing in SHAHED: (1) Building a *stock* quad tree, which is basically, a template that will be used for all quad trees of the same resolution. This one-time process is done at the system start up and used throughout the system afterward, (2) Building one daily spatial index for a new snapshot added to the LP DAAC archive by NASA. This component is called on a daily basis to add any newly added snapshots, and (3) Building spatial indexes for larger time intervals (e.g., monthly or yearly), which is done by merging smaller indexes (e.g., daily or monthly). This component is called regularly by the end of each month or year according to the temporal resolution. In the rest of this section, we describe each of these three components in details.

A. Building a Stock Quad Tree

One way to build the desired spatio-temporal index structure over existing and newly arriving data is to scan the whole data set; for each daily data snapshot, we (1) partition the data in a grid structure, (2) compute the Z-order of each point, (3) sort the points according to their Z-order value, and (4) construct the aggregate quad tree. The main overhead here is in constructing the aggregate quad tree, as partitioning is already done within the data archive itself as described in Section II, while computing the Z-order of a point is straightforward. However, we aim to make the index construction process more space and time efficient by making use of the following two properties of the data stored in tile files: (1) In each tile, the points are uniformly distributed in the space covered by the tile, and (2) There are only three available tile sizes, 1200, 2400, and 4800. These two properties make the quad trees constructed on these files share a lot of similarities, which allows us to construct *stock quad trees* and reuse them to save space and time while indexing. In a one-time offline phase, three stock quad trees are constructed for tiles of sizes 1200, 2400, and 4800. Later on, these stock trees are used while constructing and querying tiles. Below, we describe the structure of the stock trees, their construction process, and how they are used to index tiles.

SHAHED constructs one quad tree for each of the three resolutions supported by NASA, 1200, 2400 and 4800. These quad trees are built at system start up and are kept in main memory to use while building a quad tree index for a tile or while querying one of the constructed indexes. To build a stock quad tree of a specified resolution (e.g., $res = 1200$), we start with a two-dimensional array of size $res \times res$. The values in this array are not relevant to indexing as they represent non-spatial values such as temperature. Each value is assigned a coordinate (x, y) equal to its position in the two-dimensional array. This means that both x and y are integers in the range $[0, res)$. Using array position as coordinates has two advantages. First, it makes it easier and straight forward to compute Z-order values by interleaving bits from the two integer values. Second, it generates exactly the same structure for all tiles with the same resolution regardless of its position on the map which allows us to reuse the stock quad tree for all those tiles. Once coordinates are assigned to points, their respective Z-order values are computed and the points are sorted. While sorting the points, we keep track of the final position in the sorted list for each entry in the original array. This mapping is kept in a *lookup table*, which helps us later while indexing actual data by mapping each value directly to its position in the sorted array without repeating the computation of Z-order values or the sort algorithm.

Once the points are sorted, the quad tree is constructed based on the sorted order as described in [14]. Each node in the quad tree is assigned a unique ID, $start$ and end positions. The $start$ and end positions specify the range of values in the sorted order covered by this node. By the properties of the Z-curve, points covered by any node in the quad tree are contiguous in the sorted order. We start by creating the root node of the tree with $(ID = 1, start = 0, end = res \times res)$. Under the root node, four children nodes are created by partitioning the space into four quarters. The range of values covered by the root node is divided into four partitions, one for each child node. These partitions are found by detecting where the two high-order bits of Z-order values change from $00 \rightarrow 01 \rightarrow 10 \rightarrow 11$. The split process is repeated for each node as long as number of records in the node is larger than the capacity of a leaf node (e.g., 100). When four child nodes are created, they are assigned the IDs $PID \times 4 + i$, where PID is the ID of the parent node and $i \in \{0, 1, 2, 3\}$ is the child number. More details of the quad tree construction from Z-curve ordered values are in [14]. Notice that the stock tree does not hold any actual values of the input array as these values are stored in the actual quad tree for each indexed file. The stock quad tree contains mainly the structure of the tree as well as the *lookup table* created while sorting the points by Z-order values.

B. Building a Daily Spatial Index Structure

By the end of each day (at midnight), a background process is triggered by SHAHED to collect all newly inserted snapshots from the LP DAAC archive and build a spatial index for each one. Since the data in the archive is already partitioned using the sinusoidal grid, the process is simplified to building an aggregate quad tree for each tile. In this step, a tile of a standard resolution needs to be indexed. As these indexes are independent, this step can be done in parallel, where each tile is processed by a different machine.

The input is a two-dimensional array V of size $res \times res$, where res is the resolution of the tile (e.g., 1200) and the output is an aggregate quad tree that indexes the input values V. The quad tree is initialized with a header that contains two values res and c, where res is the resolution of the tree and c is the cardinality of the tree, which indicates the number of values stored at each location. For a newly constructed quad tree from a daily snapshot, c is set to one. When multiple quad trees are merged, c is updated to reflect the number of values at each point as described later in the merge process. To build the quad tree, we first sort these points in V according to their Z-order values. Instead of recomputing the Z-order values and

sorting them, the *lookup table* of the *stock quad tree* with the same resolution is fetched and used to map each point to its position in the sorted values. Sorted values are stored in a one-dimensional array V' of size res^2. The sorted array is filled in linear time by mapping each value from V to its position in V' directly using the lookup table. Once the sorted array is ready, the tree can be queried using the structure stored in the stock quad tree, which is shared among all quad trees of the same resolution. Since the stock quad tree is kept in main memory, and the size of the sorted array is the same as the original array, this quad tree index (without aggregate values) is considered a zero-overhead index in terms of storage. It is also very efficient because the quad tree structure is kept in main memory.

After the values are sorted, the next step is to compute the aggregate values in each node of the quad tree. Aggregate values are stored in a hash table in the quad tree using node ID as the key. To fill in this hash table, we traverse the tree in a bottom-up manner starting with leaf nodes. For each leaf node, the set of values stored under this node are scanned by obtaining the $start$ and end positions from the stock quad tree and iterating over them in the sorted list V'. All supported aggregate functions are calculated in a linear time and stored in the hash table. For non-leaf nodes, the aggregate values of the four child nodes are obtained from the hash table and are further aggregated to compute the aggregate values of the parent node. This procedure is repeated until the root node is reached. With the addition of the aggregate values, the overhead of the index is no longer zero but is still minimal compared to a fully structured quad tree.

C. Temporally Merging Spatial Index Structures

The constructed daily indexes are efficient for answering selection and aggregate queries on a specific day or a small range of few days. However, answering a selection query over a period of one year would still be inefficient as it requires searching 365 quad trees. To overcome this challenge, SHAHED regularly combines these daily trees into larger trees, where each tree covers a whole month or a year. This process is triggered at regular intervals (i.e., monthly or yearly) and it merges smaller trees to build a larger tree. For example, at the end of each month, this process is triggered to combine all daily indexes constructed for that month to form one quad tree that covers the whole month.

The input of the merge step is a list of quad trees of the same space (e.g., 1200×1200) and time (e.g., daily) resolution, while the output is one quad tree of the same space resolution, but lower time resolution (i.e., larger time interval) that includes all values in all input trees. The structure of the merged tree is the same as the input trees. This makes it easier when further merging the output tree (e.g., monthly) into even larger trees (e.g., yearly) using the same merge algorithm. The header of the merged tree is initialized with a space resolution res equals to the resolution of the input trees and a cardinality c equal to the sum of the cardinality of all input trees. The sorted values V' of the output tree is formed by merging the values of all input trees while keeping them in time order. This step is simple because all input lists V' are already sorted and they are all of the same size. Simply, we go over all trees in a round-robin fashion and grab one value out of the sorted values V' of each tree and store it in the output. This is repeated until all values from all trees are consumed. Notice that the memory footprint of this algorithm is minimal as the merge step can be done directly within the external storage. In case an input tree has a cardinality $c > 1$, each iteration reads c values from this tree instead of one value to keep the resulting values sorted temporally. For example, if an input tree represents a 30-day month, the 30 values are read as one block and written to output in this order.

After the sorted list V' is calculated, the final step is to compute the aggregate values in the output tree nodes. Notice that although the output tree contains more values, it has the same number of nodes with the same structure as all input trees. Think of it as another tree with the same number of records, where each record contains a list of values instead of one value. The query processor uses the cardinality c to determine the number of values in each record. Having the same number of nodes with the same structure simplifies the calculation of the aggregate values in the output tree. The aggregate values of a node in the output tree is calculated by aggregating all values in the corresponding nodes with the same ID in all input nodes.

VI. Query Processing

The spatio-temporal index introduced in SHAHED supports two types of queries, *selection* and *aggregate* queries. In selection queries, a set of values are returned in a given spatio-temporal range, while in aggregate queries, only aggregate values (e.g., average) are returned for the selected range. To provide an interactive query answer and avoid MapReduce overhead, both queries run on a single machine without MapReduce.

A. Selection Queries

In spatio-temporal selection queries, the input is a spatial rectangular range and a temporal range of dates; the answer is all readings in the specified range. For example, *find all temperature values in Minneapolis area from Feb., 10, to March 15, 2013*. The query processing runs in three steps, *temporal filter*, *spatial filter*, and *spatial refine*. (1) In the *temporal filter* step, the temporal index with the lowest granularity (i.e., year) is visited first, and if a partition in that level is completely contained in the specified temporal range, this partition is added to the selection list and the temporal range is updated to exclude the selected partitions. This process is then repeated on levels with higher granularity until the level with the highest granularity is visited (i.e., daily) which is guaranteed to cover any remaining parts in the temporal range. (2) In the *spatial filter* step, the grid in each temporal partition is used to select grid tiles that overlap the spatial range. Tiles that are completely contained in the query range are directly copied to output without further processing as all values in

them are in the answer, while partially overlapping tiles are further processed in the next step. Notice that the same grid is used in all temporal partitions which allows us to run this step once on one grid and reuse the answer with all other temporal partitions selected by the first step. (3) The *spatial refine* step processes tiles that partially overlap query range to select values that are inside the query range. Since each tile is indexed using a quad tree, the quad tree is processed to select points that satisfy the spatial range. Notice that no temporal filtering is required because we only match temporal partitions that are completely covered by the query range. No partially overlapping partitions are ever selected.

To process the range query on the quad tree, we first transform the query range from the latitude-longitude space to the sinusoidal space in which quad trees are created, and then run the range query on them. First, we apply the sinusoidal projection to each dimension in the query range to transform it to the sinusoidal space. Then, for each tile matched by the spatial filter, the query range is clipped to the range covered by this tile so that the clipped range is completely contained in the tile. The clipped query range is then normalized to the resolution of the tile such that the coordinates of the query range are integers in the range $[0, res]$ where res is the resolution of the tile, $res \in \{1200, 2400, 4800\}$. This normalization is done to transform the query range to the space of the quad tree as all points in the quad tree have coordinates in the range $[0, res]$. Finally, the stock quad tree of the matching resolution is processed with a traditional range query starting at the root and going deeper in the tree as needed. At each node, if the minimum bounding rectangle (MBR) of this node is completely contained in the query range, all values under this node are returned. If the MBR of the node partially overlaps query range and it is an internal node, the four child nodes under this node are visited and their MBRs are tested in the same way. Otherwise, if it partially overlaps the query range and it is a leaf node, all points under this node are scanned and only those in the query range are returned.

To retrieve all values in a node, the $start$ and end positions for this node are retrieved from the stock quad tree and the values in the range $[c \times start, c \times end)$ in the sorted values V' in the tree are retrieved, where c is the cardinality of the tree. Notice that all spatial attributes are kept in the stock quad tree which is completely stored in memory while only the non-spatial values (e.g., temperature) are stored in the aggregate quad tree on disk. This means that the range query is entirely executed in the main memory and only matching values are retrieved from disk which makes this range query algorithm optimal in terms of the amount of data read from disk.

Since all temporal partitions selected by the *temporal filter* step are indexed using the same grid and quad trees, the result of a range query search can be reused in all temporal partitions. In other words, the spatial range query is executed only once on one temporal partition, and when matching values are to be retrieved from disk on a particular tile, they are retrieved from all quad trees built on the same spatial tile on all selected temporal partitions.

B. Aggregate Queries

Similar to selection queries, in aggregate queries, the user specifies a spatial and temporal range; the answer is all aggregate values supported by the index for data points satisfying the spatio-temporal range. A straightforward implementation for this query is to run it as a post processing step after the selection query. However, we apply a more efficient query processing technique that makes use of the aggregate values stored in the quad tree nodes. The query runs in three steps, namely, *temporal filtering*, *spatial filtering* and *aggregate calculation*. The first two steps are the same as the selection query except for one difference. In *spatial filtering* step, all tiles overlapping the query range are sent for further processing in the aggregate calculation step. In other words, tiles that are completely contained in query range are treated the same as partially overlapping tiles.

Then, in the *aggregate calculation* step, the aggregate quad tree in each selected tile is processed to compute part of the aggregate value. For each quad tree, the query range is first normalized as described in selection queries where range query dimensions are in the range $[0, res]$. The processing starts from the root of the corresponding stock quad tree. If a node is completely contained in the query range, the aggregate values of its contents are retrieved from the corresponding node in the matching tree and accumulated to the result. Otherwise, if a node partially overlaps the query range, its four children nodes are checked. This process is repeated until leaf nodes are reached. The points under a matching leaf node are scanned and the values of points contained in the query range are accumulated. This algorithm is much faster than retrieving all points in the range as the aggregate values of trees or nodes completely contained in the query range are directly retrieved without scanning the points stored in it.

VII. Visualization

The values returned by spatio-temporal selection and aggregate queries need to be visualized for overview, analysis, and comparisons. This section describes how the results of the queries are visualized as heat maps. SHAHED supports three output formats, *static images* that represent a heat map of a selected dataset on a user-specified date, *videos* that visualize the changes of a dataset over a date range specified by the user, and *multi-level images* which represent a heat map for a specified date at different zoom levels allowing the user to navigate using pan, zoom, and fly-to interactions. The generation of both static images and videos is described in Section VII-A while multi-level images are described in Section VII-B.

A. Heat Map Images and Videos

Figure 8 shows an example of a heat map for temperature on a selected date generated by SHAHED and visualized on Google Earth. SHAHED can also generate a sequence of images where each image represents a heat map of a day in a selected date range. This sequence can be combined in a video

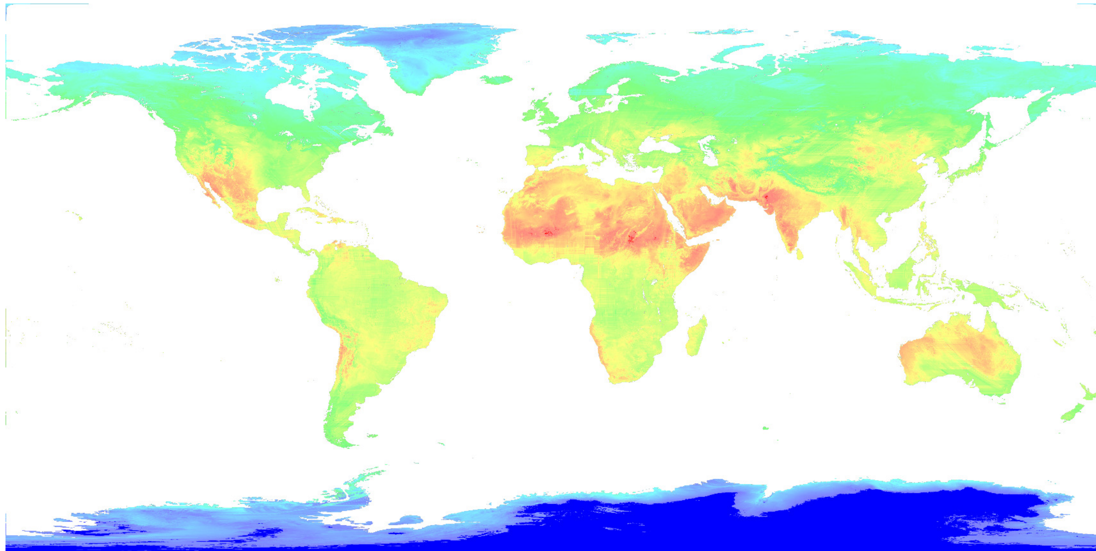


Fig. 7. A heat map of temperature in the whole world generated by SHAHED

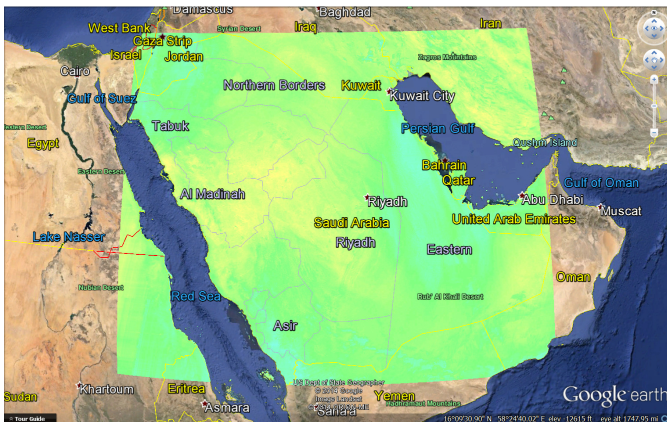


Fig. 8. Heat map of temperature viewed on Google Earth

to show the change of values over time [1]. The generated heat maps give an overall picture of value distribution and can be included in a report or a presentation. A heat map is generated as a static image using a MapReduce program described below. A video is generated as a sequence of images each representing a heat map on each day and then these images are combined to make a video.

The heat map visualization operation takes as input a dataset, a specified date, a spatial range as a rectangle, and a size of generated image as width and height in pixels. The output is an image of the specified size, which represents the heatmap for the specified area. The visualization operation works in three steps, *selection*, *tile draw*, and *overlay* steps. In the *selection* step, the archive is accessed to select grid cells that overlap with the query area. Each tile is assigned to a machine which becomes responsible of drawing this part of the heat map in the second step. In the *tile draw* step, each machine takes a tile and generates a heat map for the data in this tile. It starts by creating an image of size $width \times height$ pixels initialized with a transparent background. Then, points are read one-by-one and each point is plotted as a rectangle that represents the area it covers according to the resolution of the data (e.g., $1km \times 1km$). The color of the rectangle is selected from the spectrum of all colors according to the value of the associated point where the minimum possible value is colored blue and the maximum possible value is colored red. If multiple points map to the same pixel in the generated image, the average of their values is taken to smooth the image.

[1]Please refer to an example at http://youtu.be/hHrOSVAaak8

If the *recover* option is enabled in the uncertainty module (Section IV), missing values are automatically recovered as the input files are read so that the image becomes complete. The output of the *tile draw* step is a set of images all of the same size and each one representing part of the heat map for one tile. Finally, the *overlay* step overlays all the generated images on top of each other to generate the final picture. Since each machine plots part of the image and leaves other parts transparent, overlaying images on top of each other will generate the correct final picture. Along with the generated image, SHAHED also produces a KML file which allows the heat map to be displayed in a GIS software such as Google Earth as depicted in Figure 8. For video generation, multiple MapReduce jobs are executed by SHAHED each corresponding to one day in the range. Upon completion of all these jobs, a final call to a video generator is made to combine all images together in one video.

Figure 7 shows an example of a heat map of the temperature on April 8th, 2014 for the whole world generated from more than 300 files containing around 450 Million points. The resolution of this image is about 8000×4000 pixels and it took around five minutes to generate on a cluster of four nodes. All missing data is recovered in this image to give a smooth image that covers all land areas.

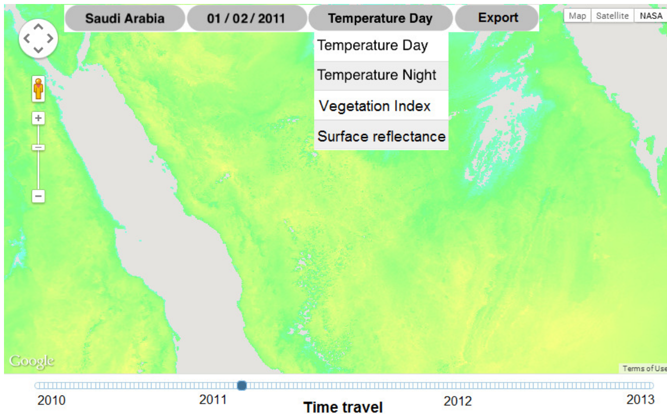


Fig. 9. A multi-level heat map displayed on Google Maps

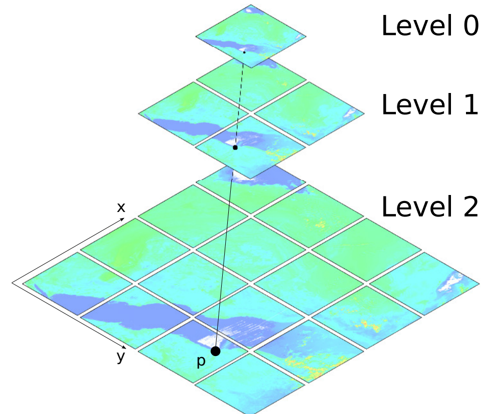


Fig. 10. Pyramid of images generated for interactive heat map

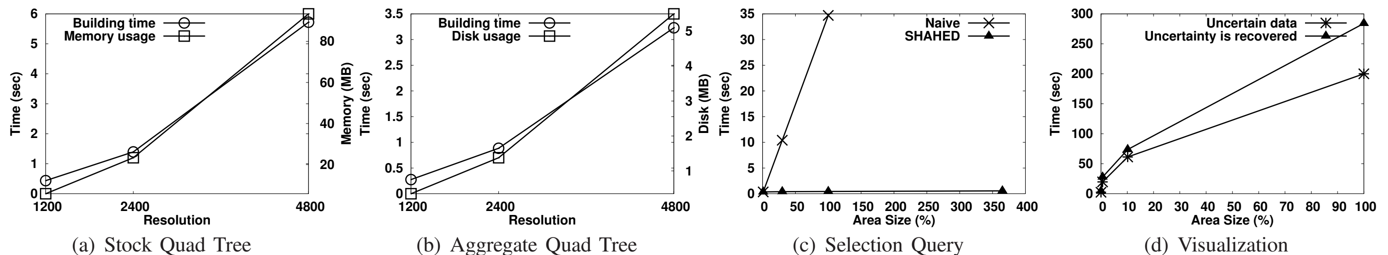
B. Multi-level Heat Maps

In addition to generating heat maps as images or videos, SHAHED is also capable of generating multi-level interactive heat maps where users can navigate through a map of the world and visualize the heat map of the visible area interactively using the standard navigation options, pan, zoom, and fly-to. Figure 9 shows an example of a multi-level heat map displayed as a layer on top of Google Maps. The main idea behind interactive heat maps is to precompute the heat maps for all regions and zoom levels. As the user navigates through the map, the system just picks from these precomputed heat map images and display them on the map. The challenge here is to generate all these images efficiently using MapReduce.

The multi-level heat map operation takes as input a dataset, a specific date, a spatial range, and number of zoom levels as integer value. The output is a set of images which represent heat maps at all supported zoom levels and regions in the specified spatial range. Figure 10 gives an example of a multi-level heat map generated at three zoom levels. Each image is of size 256×256 pixels and covers a different region based on its position in the pyramid. For example, the image at the top of the pyramid represents the whole range at the lowest zoom level. In level 1, the same area is represented by four images each of size 256×256 and so on. The operation runs in three steps, *selection*, *partition*, and *plot*. In the *selection* step, a spatio-temporal selection query is executed against the spatio-temporal index to retrieve all points in the user-specified range. Selected points are sent to the second step for further processing. In the *partition* step, a map function running in parallel on all machines scans the selected points and replicates each one to all overlapping pyramid tiles. Figure 10 illustrates an example where a point p is replicated to three tiles, one in each zoom level. Finally, in the *plot* step, each reducer takes a pyramid tile ID and all points in this tile and it plots an image of size 256×256 pixels which represents the heat map in the corresponding area. The heat map is generated exactly the same as described earlier in Section VII-A. The generated image is directly stored in the output folder with the naming convention `tile-i-x-y.png` where i is the zoom level, (x, y) is the position of this tile in the grid at zoom level i.

The above algorithm works fine, but it has a major drawback at higher levels of the pyramid. Since the tiles at higher levels in the pyramid cover larger regions, there will be more points replicated to these tiles. As an extreme case, the tile at the top level covers the whole space (e.g., the whole world), which means all points in the whole world will be replicated to this tile. For some queries where the selection area is very large, the set of selected points might contain several billions of points that should be plotted as a heat map by the machine that is assigned to the top of the pyramid. The processing of that tile might be prohibitively large. At the same time, this processing is unnecessary for two reasons. (1) Usually, images at higher levels of the pyramid do not have to be accurate as they just give a general picture. (2) The amount of details that a single image can contain is limited by number of pixels in it which is around $256 \times 256 \approx 64K$ pixels.

To overcome such unnecessary overhead, we introduce an optimization to the above algorithm to be more efficient without losing much of the output quality. In the partition step, instead of blindly replicating each point to all overlapping tiles, we adopt an *adaptive sampling* technique that only writes a random sample of points to higher levels of the pyramid. Each point is replicated to each tile with a probability that is calculated adaptively based on the zoom level. The goal is to make the expected number of points in each tile equal to number of pixels in the generated image. This makes the load more balanced as each tile is expected to contain the same number of points regardless of its zoom level. To accomplish this goal, a point is replicated to a tile at level i with a probability $\alpha_i = \alpha_0 . \min\{1, \frac{T^2}{|P|/4^i}\}$, where α_0 is the base sampling factor described below, T is the edge size of a tile in pixels (i.e., 256) and $|P|$ is the total number of points in the user specified spatial range which can be easily calculated since data is uniformly distributed and the spatial density is known beforehand. The term $|P|/4^i$ gives the number of points covered by one tile at level i, assuming data in uniformly distributed. The adaptive sampling factor α_0 is a system parameter that can be adjusted to increase the quality of generated heat maps for this algorithm. The default value of α_0 is one and it can be increased to increase number of sampled points. If more than one point are sampled and they map to the sample image pixel, their values are averaged to produce smoother looking images with higher quality.



(a) Stock Quad Tree (b) Aggregate Quad Tree (c) Selection Query (d) Visualization

Fig. 11. Performance Experiments

VIII. Experiments

In this section, we report the performance of SHAHED as it is running live on a small cluster of five machines (one master and four slaves). This cluster has Hadoop 1.2.1 and SpatialHadoop 2.2 deployed on it and running on Ubuntu 12.04. All machines have 16GB of memory, 2TB hard disk and a quad core processor which gives a total of 16 processing cores. Unless mentioned otherwise, we use the temperature dataset (MYD11A1 V005), which is collected daily at 1200×1200 resolution.

Figure 11(a) reports both building time and memory usage when building the stock quad trees. The system contains only three quad trees of resolutions 1200, 2400, and 4800. As shown in figure, it takes only a few seconds to build the stock quad tree and the memory consumption is at most 100MB for the largest one. Most of this processing and memory overhead are directly resulting from the computation of Z-order values and keeping the *lookup table* which maps each item to its sorted index. This overhead is paid once when the system starts up and is used to save computation and disk overhead when building aggregate quad trees.

Figure 11(b) gives the time and storage overhead for building the aggregate quad tree. The disk and time overhead shown in the figure result mainly from the computation of the aggregate functions as sorting is done in linear time and does not involve any comparisons. By contrasting the two figures, we could get rough estimates of the processing and disk savings, which result from the use of stock quad trees. For example, for building a single quad tree, we save about six seconds for computing Z-order values and sorting them, which is shown in Figure 11(a). In addition, for an aggregate quad tree of resolution 4800, storing the tree structure of the tree would cost roughly the same as storing aggregate values as it consists of a few numbers attached to each node. Given that the LP DAAC archive contains millions of tiles, the stock quad tree would save tera bytes of storage and thousands of hours of indexing time. The performance of the *merge* component, which merges a list of aggregate quad trees, is basically the time required to read the input files and write them back after merging, and is omitted for space limitation.

Figure 11(c) gives the performance of a selection query for a single point over time intervals of 1, 30, 100, and 365 days. This figure compares the performance of the naive implementation which runs directly on non-indexed HDF files, with the performance of SHAHED which uses our spatio-temporal index. It is clear that the naive solution does not scale at all because it needs to open a different file to obtain each point in the query range. The performance of SHAHED is almost constant as all the points in the answer are contained in only few files due to the spatio-temporal index which packs data in monthly and yearly indexes. Aggregate queries save even more time when compared to the naive algorithm as we make use of the precomputed aggregate values but the results are omitted for space limitation.

Figure 11(d) gives the performance of the visualization component when generating static images of heat maps at different sizes. In this figure, we vary the area size by choosing four areas that cover a city, a country, a continent, and the whole world with areas of size 0.0002%, 0.6%, 10%, and 100%, respectively. For each one, a MapReduce job is run to generate the image and the end-to-end time is measured. This figure shows the great scalability of the visualization algorithm where it generates a heat map for the whole world in about 200 seconds without using the uncertainty module. It also shows the efficiency of the uncertainty module where the overhead is less than 50% when compared to visualization. Generating a video involves generating a list of static images and combining the result with a video converter.

IX. Web Interface

SHAHED has a simple and interactive interface (depicted in Figure 12) that is easily accessible to end users from any web browser and provides access to all its functionality. The main area of the interface is occupied by a map which allows the user to easily navigate to any place either through pan and zoom or typing the place name to fly there directly. There is also a dataset selector that allows users to choose any dataset from the ones available in the NASA archives, e.g., temperature or vegetation. A temporal range can be set either by explicitly typing the start and end date or by using a slide control to set the requested time interval. Clicking the 'Overlay Data' check box adds an interactive heat map layer for the selected dataset on top of the current view, which can be navigated in a way similar to Google Maps. Finally, the user can click on one of the two 'generate' buttons to generate either an image or a video for the heat map of the selected region and time interval. In addition, the user can also choose an option to specify a spatio-temporal selection along with a dataset and the system will visualize the results as graphs for easy analysis and comparison. The details of each of these functionality is described below.

A. Spatio-temporal Queries

The user interface accepts spatio-temporal queries from the user and uses the spatio-temporal index described in Section V

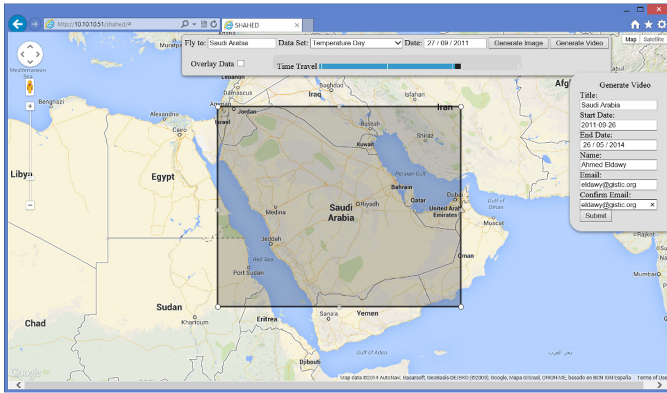


Fig. 12. Web interface

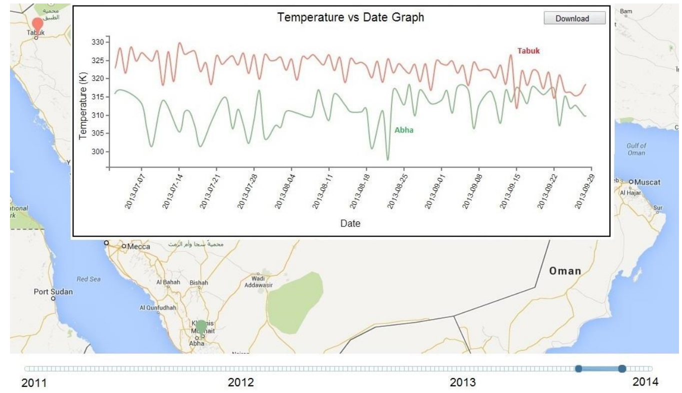


Fig. 13. Compare the temperature at two points

to answer these queries. In Figure 13, two points are selected. Then, for each selected point, a spatio-temporal selection query retrieves all values of the selected dataset (temperature) in the selected time interval and results are visualized in form of graphs. The interface also allows the user to select a range rather than a single point and an aggregate query on the selected area is executed.

B. Image/Video Generation

SHAHED provide the functionality to export a static image or a video that which represent the heat map in the specified region and time interval. For images, the user selects a dataset, specifies a region on the map, a date on the calendar, and an email address. SHAHED accepts these parameters and issues a MapReduce job in the back end which generates an image according to the specified user request. Upon completion of the job, the output is sent to the user as an email containing a link to download the requested image as both a static image and a KML file, which allows this image to be displayed on a GIS software such as Google Earth. For video generation, the user specifies all the previous information but a time interval is provided rather than one specific date. SHAHED runs a batch of MapReduce jobs, where each job generates a heat map for one day in the specified time interval. Once all jobs are completed, a call is made to a video generator to combine all generated images in a video that is finally sent to the user as a link to download on the specified email address.

C. Multi-level Heat Maps

Other than generating images and videos, SHAHED also allows users to browse the heat maps directly from the browser. Once the 'Overlay Data' check box is checked, a new layer is added to the map, which shows the interactive heat map of the current selected dataset and date. For the interactive heat map to be displayed, we precompute all heat maps of all regions and times of interest which allows the web interface to provide a smooth and interactive browsing experience for users. As the map view is changed, the browser automatically loads the set of images that cover the current view according to the zoom level and displayed region. For areas or times where the system does not have precomputed heat maps, users can still issue an export command which works with the raw data and generates the required heat map accordingly.

X. Conclusion

We presented SHAHED; a MapReduce-based system for querying, mining, and visualizing large scale satellite data. We have focused only on the querying and visualization functionality. SHAHED is composed of four main components: *Uncertainty* to recover missing data from satellite images, *indexing* that provides a novel spatio-temporal index structure for satellite data, *querying* to answer selection and aggregate spatio-temporal queries in real-time, and *visualization* that generates heat map images and videos for user queries. SHAHED is equipped with an easy-to-use web interface that grant users access to all its functionality. Reported experimental numbers from SHAHED shows it great scalability, efficiency, and usability.

References

[1] "The National Aeronautics and Space Administration," http://www.nasa.gov/.
[2] "European Space Agency," http://www.esa.int/.
[3] "MODIS Land Products Quality Assurance Tutorial: Part:1," 2012, https://lpdaac.usgs.gov/sites/default/files/public/modis/docs/MODIS_LP_QA_Tutorial-1.pdf.
[4] X. Zhou, S. Shekhar, and D. Oliver, "Discovering Persistent Change Windows in Spatiotemporal Datasets: A Summary of Results," in *BIGSPATIAL*, 2013, pp. 37–46.
[5] S. Boriah, A. Khandelwal, V. Kumar, V. Mithal, and K. Steinhaeuser, "Change Detection from Temporal Sequences of Class Labels: Application to Land Cover Change Mapping," in *SDM*, 2013, pp. 650–658.
[6] J. H. Faghmous, M. Le, M. Uluyol, V. Kumar, and S. Chatterjee, "A Parameter-Free Spatio-Temporal Pattern Mining Model to Catalog Global Ocean Dynamics," 2013, pp. 151–160.
[7] C. M. *et al*, "Climate informatics," in *Computational Intelligent Data Analysis for Sustainable Development*, T. Yu, S. Simoff, and N. Chawla, Eds. CRC Press, Apr. 2013, ch. 4, pp. 81–126.
[8] "NASA. "Global Climate Change: Vital Signs of the Planet"," http://climate.nasa.gov/.
[9] "National Science Foundation (NSF) Expeditions in Computing program. "Understanding Climate Change: A Data Driven Appraoach"," http://climatechange.cs.umn.edu/.
[10] "United States Environmental Protection Agency (EPA). "Climate Change Research"," http://www.epa.gov/research/climatescience/.
[11] "Reverb - The Next Generation Earth Science Discovery Tool," http://reverb.echo.nasa.gov/reverb/.
[12] A. Eldawy and M. F. Mokbel, "A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data," in *VLDB*, 2013, pp. 1230–1233.
[13] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, no. 2, pp. 187–260, 1984.
[14] M. Bern, D. Eppstein, and S.-H. Teng, "Parallel Construction of Quadtrees and Quality Triangulations," *IJCGA*, vol. 9, no. 6, pp. 517–532, 1999.