Indexing and Querying Spatiotemporal Raster Data

Dennis B. Andersen
buhhko@cs.aau.dk

Martin L. Kristiansen
martinlk@cs.aau.dk

Claus H. Poulsen
kludder@cs.aau.dk

Thomas Winterberg
thomasw@cs.aau.dk

Department of Computer Science, Aalborg University, Denmark

June 12, 2008

Abstract

This paper addresses the issue of indexing and querying spatiotemporal raster data. This is done by developing a prototype named WEATHR that allows users to make queries on historical precipitation data, e.g., to determine the origin of water masses that has caused a flooding. Three functionally equivalent implementations have been developed. One is based on the GeoRaster component for Oracle Database, providing a performance baseline. Another is based on storing single pixel values along with their corresponding coordinates in a table. The last one is similar to the latter, but uses the Hilbert space-filling curve for indexing two-dimensional raster space.

Among the key findings is that the Hilbert space-filling curve is not well-suited when raster data is sparse, nor when data is queried using small, non-rectangular search windows. Furthermore, the GeoRaster-based implementation has a storage requirement that is four times bigger compared to the other two, and a query performance that is slower by several orders of magnitude.

Keywords: Raster data, Hilbert space-filling curves, index-organized tables, GeoRaster, range query, k-NN query, sparse data.

1 Introduction

The WEATHR prototype, as presented in [1], is a location-based service capable of providing various services related to precipitation. The services offered by the prototype consist of monitoring user-defined regions and routes. If precipitation will occur inside or on these, respectively, the prototype is able to warn the user in advance—how far in advance is determined by a forecast algorithm. If combined with a route recognition mechanism, the service will be able to warn users traveling along a route on which precipitation is about to occur. Another example is that of a painter working outdoors. It would be very convenient for him to receive an early warning of upcoming rainfall, allowing him to stop working and the paint to dry before it actually starts raining. A scenario showing a region is illustrated in Figure 1.

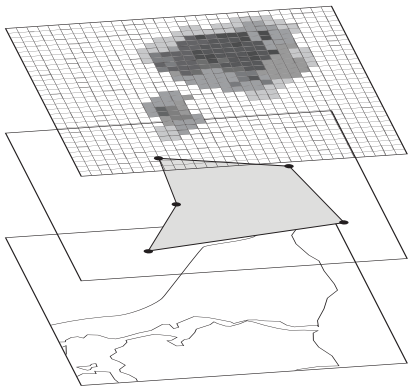


Figure 1: Layered data overview. From top to bottom: precipitation data layer, region layer, geographical reference layer.

The prototype bases its services on data provided by a Local Area Weather Radar (LAWR)—a device capable of recording the aerial distribution of rainfall within its coverage area. LAWR data sets are in raster format, where each pixel is an 8-bit byte value [0; 255]. Pixels with a value greater than zero represent an area that is currently experiencing rainfall—the greater the value, the heavier the rainfall. Figure 2 illustrates an example of a 6×6 LAWR data set (a real-world data set is typically larger). WEATHR uses data provided by a LAWR

0	0	0	0	0	0
0	0	42	35	0	0
0	0	3	57	14	0
0	0	0	2	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Figure 2: 6×6 LAWR data set.

located in North Jutland, Denmark.[1] The radar delivers a 240×240 image where each pixel represents a 500m $\times$ 500m area.

The focus of this paper is to describe an extension of the WEATHR prototype that introduces advanced query types on spatial LAWR data. In particular, all data received from the LAWR is stored in a database, allowing the prototype to perform historical queries on the entire range of stored LAWR data sets. Consider, for example, a farmer that uses WEATHR to estimate how much rainfall his land has received over a specified period of time. As another example, using WEATHR to better determine the cause of sewage overflowing or similar would also be feasible.

The contributions of this paper are summarized as follows:

- A working prototype that demonstrates three types of historical queries on LAWR data:
  - Range
  - Single-point
  - k-nearest neighbor
- A performance comparison in terms of execution time and space usage of implementations using the Hilbert space-filling curve and scanline data organization, respectively.
- A comparison of the two above-mentioned implementations to the commercial GeoRaster component for Oracle Database.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 provides an abstract overview of the basic components in the prototype and outlines the workflow involved in executing a query. Section 4 describes three functionally equivalent solutions for implementing historical queries in WEATHR. Section 5 provides an overview of the technical system architecture and details on implementation. Section 6 presents performance studies which provides performance measurements on real-world data combined with data analysis. Section 7 describes future work and Section 8 concludes the paper.

[1] The UTM coordinate of the LAWR is: Zone 32, 547636 meters east, 6318784 meters north (northern hemisphere).

2 Related Work

Storing raster data in a DBMS is supported by several database products, such as the GeoRaster component of the Oracle Spatial Cartridge [2] for Oracle Database and the Rasdaman database server [3]. Both Oracle GeoRaster and Rasdaman support georeferencing spatial raster data such as satellite or radar imagery, allowing retrieval of data by supplying coordinates on the Earth's surface.

For this project, Oracle Database has been chosen as the underlying data source for the WEATHR prototype. The reason behind this is that Oracle Database is freely available for non-commercial use and offers the Spatial and GeoRaster components. The GeoRaster component serves as a commercial off-the-shelf (COTS) reference implementation for storing and querying spatial raster data in a DBMS. However, the out-of-the-box capabilities of GeoRaster does not meet the requirements of the WEATHR prototype. GeoRaster is geared toward the efficient delivery of large images for onscreen viewing. It does not support querying arbitrarily-shaped subsets of raster data from within a single image. GeoRaster outputs only bitmap formats (such as JPEG, PNG, etc.), which are by design rectangularly dimensioned. Querying arbitrarily-shaped subsets of raster data is, however, required by WEATHR, since a user is allowed to query regions which are defined by unconstrained polygon shapes—convex as well as concave. This justifies the implementation of a custom-built GeoRaster-based solution that supports the above requirement.

Reducing data from multiple dimensions to one enables simple and well-known access methods to be utilized, such as the B^+-tree [4, 5, 6]. The way to perform such a reduction is to use a *space-filling curve*. Space-filling curves represent a technique suitable for encoding the entire range of a multidimensional space as a one-dimensional curve—i.e., a one-to-one correspondence mapping $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. A curve can be thought of as the path of a continuously moving point—a point that moves through the multi-dimensional space in an order that preserves locality in the reduced space. More formally, a curve is an indexing which is continuous with respect to the discrete topology given by a distance

metric, such as the Manhattan metric [7]. The Hilbert space-filling curve [8] is widely accepted as achieving the best locality preservation and provide significant performance benefits with regard to data access [9, 10]. This assertion is substantiated by experiments conducted by Faloutsos et al. [4, 10, 11].

As mentioned earlier, WEATHR must store all LAWR data it receives and answer historical queries based on these data. Research has been conducted on Hilbert curves for indexing multidimensional grids [7], i.e., dimension $d \geq 2$. LAWR data consists of two-dimensional raster images, but a temporal dimension may be considered the third dimension. However, indexing grids with $d = 3$ is inherently for cube space only, meaning that as the third (temporal) dimension grows, so will the other two (x and y). Locality in the reduced space is then an approximation of that in the original three-dimensional space. This property, however, is not desirable in the case of indexing multiple LAWR images, in which the temporal dimension continues to grow while the other two remain constant. Due to this, LAWR images in WEATHR are indexed using a Hilbert curve where $d = 2$. Efficient access with regards to temporal locality is ensured by a composite database index.

Space-filling curves are used in the field of databases as well as computer graphics. McCoo et al. [12] describe an algorithm that rasterizes a polygon. The algorithm rasterizes along a Hilbert curve and stores a sparse matrix in Hilbert-order. Generally, a matrix is sparse if it is primarily populated with zero-entries and contains very few non-zero elements per row [13]. Duff et al. [14] shows practical approaches to the efficient utilization of sparsity. Their idea is to store only non-zero values of sparse matrices, as opposed to storing all values. It is possible to reduce storage and computational requirements by storing only non-zero values [15, 16]. This substantiates the approach used for the first prototype of WEATHR, which used a memory-based approach, storing only non-zero values. This was based on statistics showing that approximately 97.2 percent of the values in 18 days worth of data (a total of 5,184 data sets) were zero-values [1].

A number of data access methods for spatial data exist. Among these methods, the R-tree [17] is the most popular. The R-tree is a dynamic B-tree-like data structure for representing spatial data. Spatial objects are approximated by their *minimum bounding rectangle* (MBR), which is a much simpler container than the object's original shape. Although the R-tree has lots of applications, it has some drawbacks [18, 19]. E.g., a spatial query is performed by first examining the root node of the tree and then recursively searching through all children nodes that intersect the search rectangle. A spatial query can overlap multiple MBRs and consequently a query may require several nodes to be visited before determining which objects the query intersects. The query performance is thus proportional to the number of nodes covered by the search rectangle. Also, as updates occur, the tree evolves by applying appropriate insert and delete operations, thereby discarding the previous version of the tree. Due to this, the R-tree is not applicable when storing historical LAWR data, since the tree indexes a current state and discards previous states. However, indexing multiple versions of data can be done by making the tree multi-dimensional. The Multi-Version R-Tree (MVR-tree) [20] conforms to this property and is used in, e.g., CAD systems that index multiple versions of the same drawings, where the drawings differ only slightly from one another. The idea behind the MVR-tree is to share as many common nodes as possible between versions and thereby reduce duplication. The MVR-tree is, however, not suitable for indexing raster LAWR data. Conceptually, a rain cloud moves as time progresses, making an R-tree-like indexing ideal. However, pixels in a raster image never actually move, but simply change value as time progresses. This means that it does not make sense to index moving rain clouds in a raster data set since pixels are stationary objects.

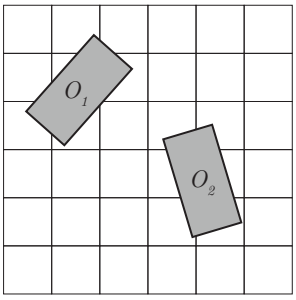


Figure 3: Objects O_1 and O_2 overlapping raster data.

Figure 3 shows two spatial objects, O_1 and O_2, overlapping a raster grid. The figure attempts to illustrate that since we only deal with raster data, and not vector-based objects such as O_1 and O_2, indexing techniques such as space-filling curves are more suitable compared to, e.g., a spatial index such as the R-tree. As described by Ooi and Tan [6], the use of space-filling curves causes the space to be partitioned into cells of uniform size where each cell is labeled with a unique number. However, a spatial object may not fit into a single cell and con-

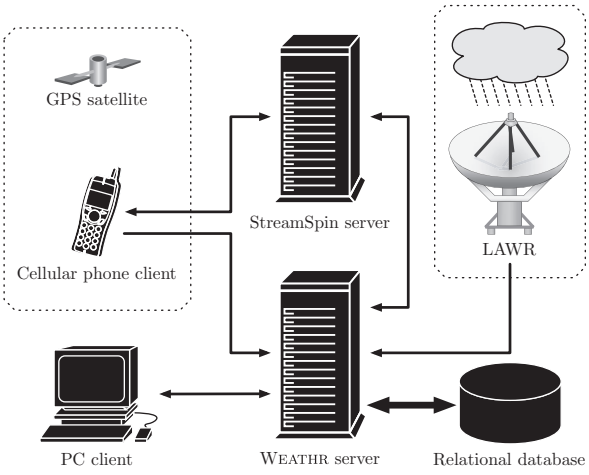


Figure 4: System overview.

sequently an object may overlap several cells. But since LAWR data sets are formatted as raster (or gridded) data, this is not a problem since it is possible to achieve a one-to-one mapping between the generated curve and the uniform, raster-formatted LAWR data.

3 System Overview

The WEATHR prototype is a system capable of querying precipitation data received from LAWRs, as described in [1]. This paper addresses an extension of the prototype, implementing historical query support by storing LAWR data in a relational database and subsequently querying it. Figure 4 illustrates the major components in WEATHR. This section briefly explains each of those components.

WEATHR is built on the client-server model. The client is either a PC running web browsing software or a cellular phone running the StreamSpin client software [21]. When using a PC, users can navigate a Google Maps interface [22], viewing the latest LAWR output as an overlay on the map. Additionally, users can create and save *areas of interest* using a point-and-click interface. Areas of interest represent areas that a user is interested in monitoring continuously. Creating such an area enables a user to view estimated forecast data for the area, as well as receive notifications of upcoming rainfall in that area on a mobile phone via the StreamSpin service. An area of interest is a polygon shape and will from here on out be referred to as a *geometry*. Figure 5 shows a screenshot of the web front-end with a geometry visible on the map. For more details on WEATHR and its capabilities, see [1].

The dashed box containing the LAWR in Figure 4 represents the weather radar delivering data to the system. Data is currently being received from a single LAWR every five minutes, but the implementation is generic enough to accept data from multiple radars and in other time intervals as well. When a data set is received at the WEATHR server, it is immediately stored in a relational database. As time goes by, the database builds up an archive of data sets that can be accessed by the system. This allows a user of the web front-end to execute historical queries by creating geometries and specifying various query properties, such as temporal range defined by query start and end time. An abstract overview of the workflow involved in executing a historical query is illustrated in Figure 6. A user first creates a geometry or selects an already created one and then specifies a set of historical query properties. The user then submits the information, sending the query to the server. On receiving, the server prepares an SQL statement equivalent to the specified query properties and executes the statement. In reality, the process is much more involved than just preparing a single SQL statement—the exact details are explained in Section 4. Finally, a report is generated from the SQL result set and sent back to the user where it is displayed. In the case of range and single-point queries, the result is displayed as a chart showing the temporal distribution of precipitation throughout the specified temporal range. In case of a k-NN query, the result is displayed as a simple summary of the k values found and their corresponding coordinates.

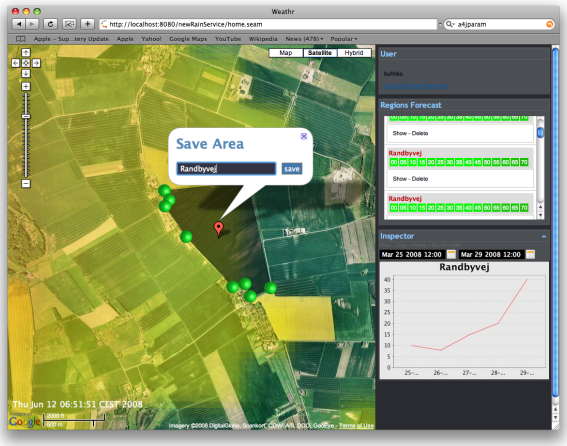


Figure 5: WEATHR web front-end screenshot.

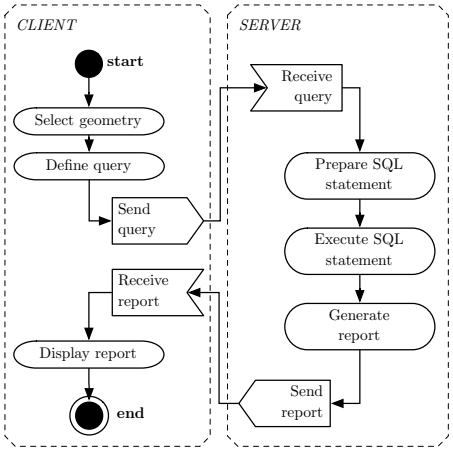


Figure 6: UML state diagram illustrating the basic workflow process of submitting, executing, and displaying historical queries.

4 Queries & Algorithms

This section describes three solutions for implementing historical query ability in WEATHR. The first two solutions have been built specifically to solve the problem at hand. They have both been implemented on top of Oracle Database and each utilize their own optimized schema design. The first of the two is a simplified approach for storing pixel values in a table—that is, one pixel value per record along with corresponding (x, y) coordinates for that particular pixel. The other is similar, only based on one-dimensional space-filling curves instead of two-dimensional coordinates. The third solution is based on the commercially available GeoRaster component for Oracle Database—a component capable of handling spatial raster data—and acts as a baseline against which to measure the performance of the two first solutions. Each solution must provide support for a number of query types:

Single-point query Allows users to query a specific position on the Earth's surface by providing geodetic coordinates (such as UTM or latitude/longitude) and a temporal range that defines the time interval from which the data must be gathered.

Range query A user-defined geometry consisting of either a convex or concave polygon is used as the basis for performing a spatiotemporal range query. This query type enables users to collect statistics about, e.g., how much rainfall a specific area has received over a period of time. A spatiotemporal range query in WEATHR has two pairs of upper and lower boundary values (i.e., ranges) that are used for retrieval: a spatial range defining pixels that are covered by a query as well as a temporal range. Figure 7(a) illustrates an example of a spatial range.

k-nearest neighbor query The final query type is capable of answering a question such as "What are the locations of the two nearest rain showers?" Imagine for example an area being flooded due to insufficient drainage along the water's travel path. A k-nearest neighbor (k-NN) query on historical data could be used to determine the origin of the water masses, enabling optimization of the drainage system.

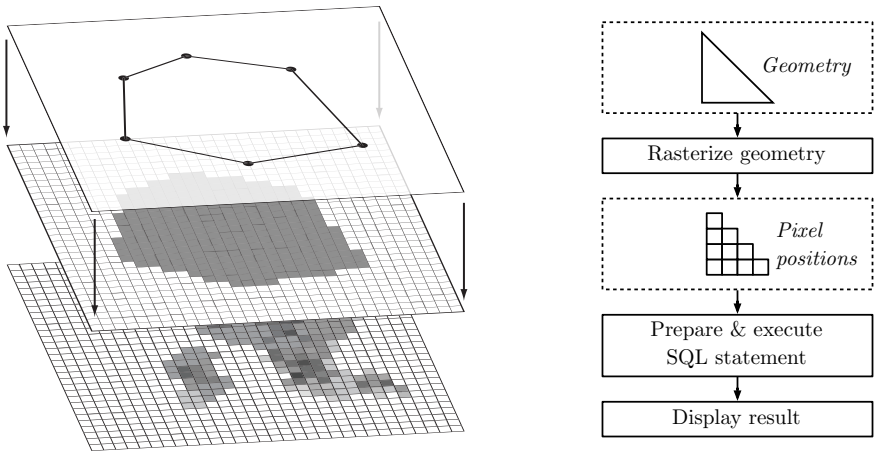
In order to obtain a high level of independence between application and database, the solutions described here have all been implemented at the database level. This ensures that the entire process of executing queries on LAWR data is totally separated from and independent of a client application, such as the WEATHR web front-end.

The rest of this section describes in detail the design and implementation of each solution. Particular care is taken to explain how each supports the query types listed above. First, however, the essential concept of geometry rasterization—a technique introduced in [1] for the initial prototype of WEATHR—is shortly reviewed.

4.1 Geometry Rasterization

When performing a spatial query using a geometry to define the spatial range, the geometry must first go through a rasterization process. The reason for this is that in order to perform a spatial join between a geometry and a LAWR raster image, the geometry must be converted to the same discrete raster space as that of the image. After the rasterization process has finished, the dimensions of both the LAWR image and the rasterized geometry are exactly the same. This means that a pixel position (x_0, y_0) in the rasterized geometry has the same spatial occupancy as a corresponding pixel located at (x_0, y_0) in the LAWR raster image.

Figure 7(a) illustrates the principle behind performing a spatial join between a geometry and a LAWR image by rasterizing the geometry. The two top layers show a geometry before and after rasterization, respectively. The bottom layer shows a LAWR raster image that has the exact same spatial extent and number of pixels as the layer above it. This enables the application to iterate through the pixel positions in a rasterized geometry while at the same time visiting the corresponding pixels in the



(a) Spatial join illustrated in three layers.

(b) Basic workflow.

Figure 7: Rasterization.

LAWR image, effectively collecting only the LAWR pixel values that overlap the original geometry.

The basic workflow of performing a query in the prototype is outlined in Figure 7(b). The rasterization process takes as an input parameter a geometry in the form of a polygon and outputs a number of pixel positions. These pixel positions are then used to construct an SQL statement that queries the pixel values at the corresponding positions in an LAWR image. It is important to note that the rasterization algorithm outputs (x, y) pairs that correspond to (x, y, v) triplets in an LAWR image, where v is the pixel value. For more details on the inner workings of the geometry rasterization algorithm, see [1].

4.2 Coordinate-Based Solution

This section describes the first solution. It is based on a relatively simple database schema for storing raster images in a relational data model. It stores one pixel value per record along with a corresponding raster space coordinate for that particular pixel.

A similar database schema, depicted in Figure 8, is used for the first two solutions. They both carry the simple concept of an **image** entity that is comprised of raster content represented by pixels—a concept which is realized as a master-detail table design. A table containing a single row per image describes the basic properties of the image while another table contains the actual raster data. The latter is named the raster data table (RDT). For the sake of simplicity, a relational normalized approach has been chosen for the two solutions not built on GeoRaster.

The database schema for the first solution is presented in Figure 9. A table **radar** contains a row for each LAWR registered in the system—in our case, we have just a single radar providing data, but multiple radars are supported throughout the application. The most important columns on this table are:

pixel_size: Length and height in meters of each pixel in an image output by the particular LAWR. In our case, each pixel is $500\text{m} \times 500\text{m}$.

cols: Number of columns in a raster image.

rows: Number of rows in a raster image.

bounds: Bounds of an image as specified by the SQL type `SDO_GEOMETRY`. In other words, this attribute georeferences an image.

Some of the columns on the **radar** table could be argued to be properties of the individual images rather than the LAWR. For instance, an LAWR can provide images in a number different resolutions, which is impossible to model using the schema design in Figure 9. In such an event, a new row for the same LAWR will have to be inserted in the **radar** table. The same is true in case an LAWR physically moves—e.g., a mobile radar towed by a car. This design has been chosen in order to minimize redundancy through normalization, requiring only four columns in the **image** table.

Each LAWR in the system has a number of images associated with it, modeled by the one-to-many referential constraint between the two relations **radar** and **image**. Important columns on the **image** table are:

retrieved_at: The time at which an image was received from the LAWR.

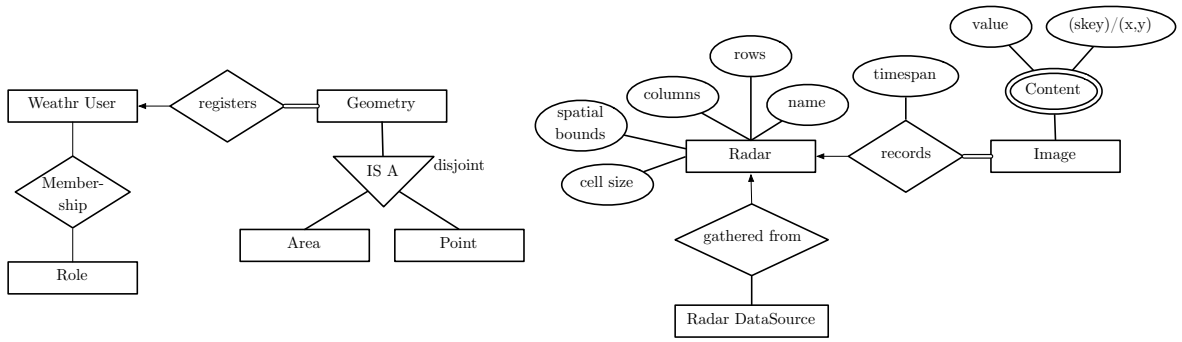


Figure 8: ER diagram of the overall database schema with essential attributes only.

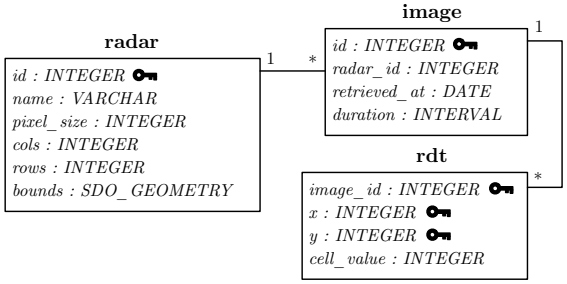


Figure 9: Coordinate-based schema.

duration: The temporal extent of an image. The duration is measured back in time, starting from the *retrieved_at* attribute. In our case, each image has a duration of five minutes.

As mentioned, the actual raster data is stored in a separate RDT, named **rdt** in Figure 9, where each row is a tuple $(image\_id, x, y, cell\_value)$. Each row in the **image** table has a number of associated rows containing pixel values in the **rdt** table, modeled by yet another one-to-many referential constraint. Due to each image consisting of 240×240 pixels, the amount of rows required per image is 57600. However, as discussed in Section 2, a high compression ratio can be obtained by storing only non-zero values. By storing meta values that describe the dimensions of the original image in the **radar** table (i.e., columns *rows* and *cols*), the compression is in effect lossless since we are able to derive the discarded zero values.

4.2.1 Indexing

For optimal access patterns when querying data, appropriate indexing schemes have been carefully chosen. A property of the **rdt** table is that it has only four columns, each containing a simple `INTEGER` type. Oracle tables are per default heap tables, providing no guarantee on order. Any index applied on top of a heap table will thus be dense. This combined with the fact that the RDT contains a massive amount of entries, any index applied on top will cause a significant space overhead compared to the actual size of the data itself. Comparing segment sizes of a populated RDT with default 90% block filling (PCTFREE 10%) and the primary key index shows almost identical allocation sizes for the two, which means the index represents an overhead the size of the data itself.

This led to the choice of using Oracle-specific index-organized tables (IOTs) [23]. An IOT is a concept where table and index are merged together. Conceptually, an IOT is a B^+-tree variant where data is stored inside the leafs of the tree and nodes are arranged by the logical primary key [24]. This guarantees that data always be stored physically in search key order based on the primary key. This boils queries down to a single index scan which minimizes block reads since it is no longer necessary to retrieve native row identifiers (*rowids*) from the index and subsequently fetch the data from somewhere else on disk.

Performance measurements documented in [25] show that significant time savings on queries and updates can be obtained by using IOTs on the expense of higher cost of delete operations. The **rdt** table has only four columns, and the data herein is never deleted nor updated. Given these two properties, tests in [25] show only a minor impact on inserts, yet significant time savings present themselves when querying the primary key. This suggests that the **rdt** table will benefit significantly from being implemented as an IOT instead of a regular heap table with an indexed primary key column. The **rdt** table is modeled as a weak entity which involves a composite primary key definition composed of the primary key of the **image** table

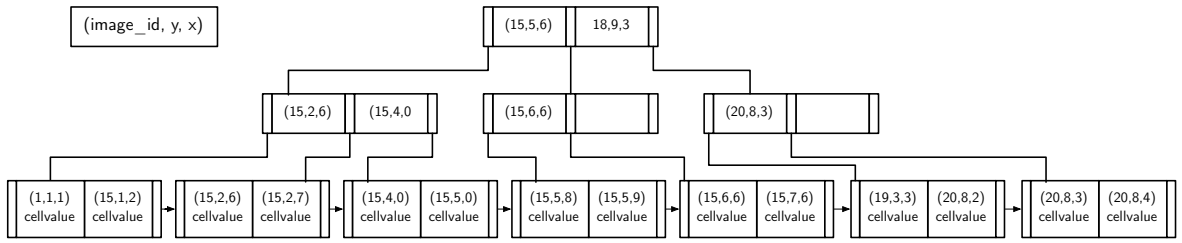


Figure 10: Index tree for the coordinate-based solution.

and the two columns x and y. Figure 10 illustrates the **rdt** table as an IOT.

The primary key of the **image** table is a computed sequence. Although the primary key is artificially generated, the sequence reflects a sense of logical time since the table is append-only and data is inserted chronologically as it is recorded by the LAWR. A spatiotemporal query involves joining the **image** table and the **rdt** table. Queries on the **image** table will rarely be conducted on the primary key, making the use of an IOT unwise. Furthermore, the row count of the **image** table will always be several magnitudes lower than that of the **rdt** table, meaning that it will not represent a performance bottleneck. All queries on the **image** table involve querying the *retrieved_at* column, either by executing a single-point query or, perhaps more interesting, a range query. Due to all these properties, a standard heap table has been chosen for the **image** table with a standard Oracle B^+-tree index applied on the *retrieved_at* column.

```
1  coordRange(geom:SDO_GEOMETRY; t1,t2:
       TIME_TYPE):SQL_RESULT is
2   do
3    pixels:=rasterize(geom);
4    sl:=toScanLines(pixels);
5    sql:=buildQueryString(sl,t1,t2);
6    return dbExecute(sql);
7   end;
8
9  toScanLines(pixels:ARRAY):LIST is
10  do
11   sort(pixels);
12   lineStart:=pixels[0];
13   lastRead:=pixels[0];
14   for i:=1 to pixels.length-1 do
15    if (!hAdjacent(lastRead,pixels[i]))
16     lines.addLine(lineStart,lastRead);
17     lineStart:=pixels[i];
18    end;
19    lastRead := pixels[i];
20   end;
21   lines.addLine(lineStart,pixels[pixels.
         length-1]);
22   return lines;
23  end;
```

Listing 1: Range query for coordinate-based solution.

4.2.2 Range Query

A spatiotemporal range query facilitates retrieval of pixel data from within a spatial range as well as a temporal range. A spatial range is defined by a user-created geometry. A geometry has spatial extent, consisting of a set of points that together form a polygon shape. A temporal range is simply an interval in time, consisting of a start and end time.

The process is as follows. The user draws a geometry using the Google Maps interface and saves it in the system. He then specifies a temporal range using controls available on the Weathr web frontend. This information is then sent to the server where it is passed on to the underlying database for processing. On receive, the geometry is located in a table **geometry** where it is stored as an `SDO_GEOMETRY`—see Figure 8 for a conceptual ER diagram. This geometry is then filtered through the rasterization algorithm where it is converted to a set of pixel positions. After this, the pixel positions are sorted in a horizontal scan line fashion, separating each scan line. Finally, an SQL statement that executes the range query can be generated.

Listing 1 shows the algorithm for executing a spatiotemporal range query. The function `coordRange` in line 1 is the entry point. It has three input parameters. The first is a geometry `geom` defining the spatial range. The last two are timestamps `t1` and `t2` defining the temporal range. In line 3 the geometry is rasterized, where output is an array of pixels. Line 4 calls a function `toScanLines` that generates the scan lines necessary for building the final SQL query. In this function, line 11 sorts the array of pixel positions with regards to both y and x in an ascending order, where y precedes x. This establishes a scan line order on the `pixels` array. Lines 12–21 separate each scan line in the sense that a line is a number of pixels that all have the same y-

$$
\begin{aligned}
\text{join} &\leftarrow image \bowtie_{image.id=rdt.image\_id} rdt \\
\text{temp}_0 &\leftarrow \sigma_{t \geq t_1 \wedge t \leq t_2}(\text{join}) \\
\text{temp}_1 &\leftarrow (\sigma_{y=y_0 \wedge x \geq x_0 \wedge x \leq j_0}(\text{temp}_0)) \cup \\
&\quad (\sigma_{y=y_1 \wedge x \geq x_1 \wedge x \leq j_1}(\text{temp}_0)) \cup \\
&\quad \vdots \\
&\quad (\sigma_{y=y_n \wedge x \geq x_n \wedge x \leq j_n}(\text{temp}_0)) \\
\text{result} &\leftarrow \Pi_{t,t\mathcal{G}_{\mathbf{sum(v)}}}(\text{temp}_1)
\end{aligned}
$$

Query 1: Relational algebra statements for performing a coordinate-based range query.

$$
\begin{aligned}
\text{join} &\leftarrow image \bowtie_{image.id=rdt.image\_id} rdt \\
\text{temp}_0 &\leftarrow \sigma_{t \geq t_1 \wedge t \leq t_2}(\text{join}) \\
\text{temp}_1 &\leftarrow \sigma_{x=x_0 \wedge y=y_0}(\text{temp}_0) \\
\text{result} &\leftarrow \Pi_{t,v}(\text{temp}_1)
\end{aligned}
$$

Query 2: Relational algebra statements for performing a coordinate-based single-point query.

coordinate and adjacent x-coordinates. Note that the function $hAdjacent(p_0, p_1)$ returns true for two pixels p_0 and p_1 that are located directly next to each other on a horizontal line—e.g., $(2, 3)$ and $(3, 3)$ but not $(2, 3)$ and $(4, 3)$. Once all scan lines have been determined, the `toScanLines` terminates and control is returned to the `coordRange` function. Line 5 calls a function `buildQueryString` that builds an SQL string equivalent to the relational algebra statements in Query 1. It takes a list of n scan lines $\langle l_0, l_1, \ldots, l_n \rangle$ and two timestamps t_1 and t_2 as inputs. Each scan line is a pair of coordinates, i.e., $l_n = ((x_n, y_n), (j_n, y_n))$. Notice that both coordinates always have the same y-coordinate due to them being endpoints of the same horizontal scan line. First in Query 1, all image data is assigned to the join variable. Second, the number of rows is significantly reduced by applying a predicate that selects the temporal range defined by t_1 and t_2. This is assigned to temp_0. Third, the expression being assigned to temp_1 expands to n predicates—one for each scan line. This selects the spatial range into temp_1. Finally, a projection using a `GROUP BY` statement and the aggregate function *sum* boils the result down to one row (t, v_{sum}) per time slot, where v_{sum} is the summed precipitation amount for the entire spatial range. After the range query has been executed the result is returned in line 6 of Listing 1.

4.2.3 Single-Point Query

A single-point query is the simplest query type. It basically consists of mapping a real-world coordinate to a coordinate in raster space, followed by a lookup in the database.

A single-point query can be considered the equivalent of a range query based on a polygon that covers a single pixel only. The rasterizer component is capable of taking a real-world coordinate (e.g., latitude/longitude) as a parameter instead of a polygon. This causes the output to be a single pixel position only. This pixel position is the (x, y) pair that corresponds to the (x, y, v) triplet that is georeferenced so that it contains the input real-world coordinate. Once a pixel position has been determined, a query equivalent to that in Query 2 is executed, where t_1 and t_2 define the temporal range and (x_0, y_0) is the position of the pixel value being looked up.

4.2.4 k-NN Query

A k-NN query in WEATHR is basically a spatiotemporal range query with a dynamic spatial range. The query returns the k nearest pixels where precipitation has occurred within a specified temporal range.

The process of executing this query type consists of creating a number of increasingly larger "ring" images where each pixel has a value that is the sum of the corresponding pixel values from all original LAWR images within the temporal range. Consider Figure 11. The center pixel O is the point of origin—or starting point—of the algorithm. If O contains precipitation it will be added to the result set. If not, it will not be added. After examining O, the algorithm continues outward, looking for pixels containing precipitation. First, the pixels covered by the ring L_1 will be examined. After L_1, ring L_2 is examined, followed by L_3 and so on. When k or more pixels have been found, the algorithm terminates. This means that a k-NN query guarantees returning a *minimum* of k results. In other words, all results from the last visited ring will be included in the result.

The algorithm for a k-NN query is presented in Listing 2. As can be seen in line 1, the input parameters consist of a pixel position determining the starting point of the algorithm, a pair of timestamps determining the temporal range from which to gather data, and a value k determining the minimum number of results to be returned. Line 6

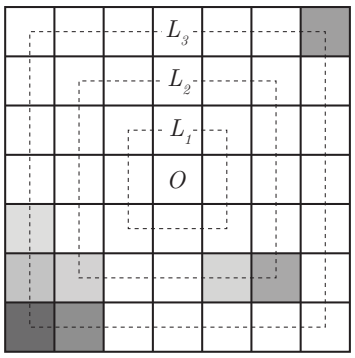


Figure 11: k-NN rings. Gray pixels contain precipitation.

```
1  coordkNN(center:COORD_TYPE; t1,t2:TIME_TYPE;
       k:INT):SQL_RESULT is
2  do
3    running:=true;
4    kFound:=0;
5    ringLevel:=0;
6    while (running) do
7      ring:=getRing(ringLevel,center);
8      sl:=toScanLines(ring);
9      sql:=buildQueryString(sl,t1,t2);
10     subResult:=dbExecute(sql);
11     kNNResult.union(subResult);
12     kFound:=kFound+subResult.size();
13     if(kFound ≥ k)
14       running:=false;
15     end;
16     ringLevel++;
17   end;
18   return kNNResult;
19 end;
```

Listing 2: k-NN query for coordinate-based solution.

contains a while loop that runs until k or more results have been found and the algorithm can terminate. In lines 7–8 a ring of pixel positions is calculated and converted to a list of scan lines using the `toScanLines` function from Listing 1. Line 9 builds an SQL expression very similar to that of the range query—see Query 1. The only difference, expressed in relational algebra, is the projection:

$$\text{result} \leftarrow \Pi_{x,y,(x,y)\mathcal{G}\mathbf{sum(v)}}(\text{temp}_n)$$

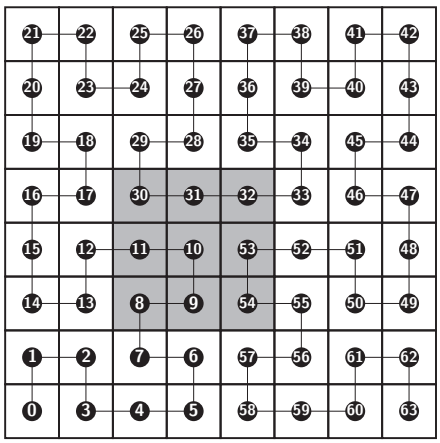
The result of the projection is a row per pixel (x, y, v_{sum}), where v_{sum} is the sum of each individual pixel within the specified temporal range. A convenient consequence of storing only non-zero values in the database is that only pixels containing non-zero values are returned, making it easy to identify results and as well as determine how many there are. This takes place in lines 11–12. Note that the `union` method on a result set in line 11 simply adds two result sets together. Lines 13–15 check if k or more results have been found. If that is the case, the algorithm terminates and returns the result.

The actual implementation of the k-NN query contains an additional conditional check. This consists of a user being able to define minimum and maximum thresholds for pixel values, such that the algorithm will ignore values that do not lie within a certain range. This has, however, been left out of Listing 2 due to brevity.

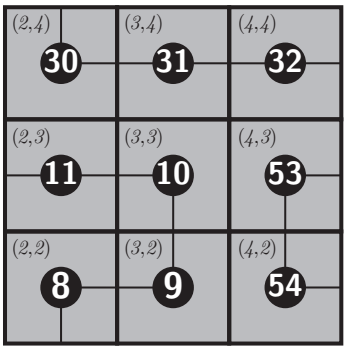
4.3 Hilbert-Based Solution

The idea behind space-filling curves is to map multi-dimensional data into one dimension while preserving data locality during the mapping. A curve is a continuous function that lies in, e.g., the two-dimensional plane (a so-called *plane curve*). The Hilbert curve, for example, is continuous with respect to the discrete topology given by the Manhattan metric[7]—see Figure 12(a). The original space is divided into a number of cells and each cell is labeled with a unique scan index number, referred to as an *skey*. The cells are labeled in an increasing order according to the path of the space-filling curve. The division must conform to certain properties[11]: all cells must have equal size and there must be no overlap between cells—essentially, the whole space is divided into a grid. An advantage of this mapping is the possibility to use well-known indexing methods for a one-dimensional space, such as the B^+-tree.

As already mentioned in Section 2, the Hilbert curve has the best locality-preserving behavior compared to other space-filling curves, which can be taken advantage of when querying a spatial region. Consider a B^+-tree on a column of *skey* values. Given that a geometry covers a set of pixels that are spatially close to one another, the index scan will ideally be very efficient since the index will be favorably organized, having a locality-preserving order on the leafs in the tree. The Hilbert curve divides each side of a square into two equal size parts, which divides the square into four smaller equal size squares. Each of these squares is similarly divided into four smaller squares etc. The Hilbert curve is therefore only applicable to spaces where each side length is equal to an integer power of two. However, as described in Section 1, we receive LAWR raster images with dimensions 240×240, which does not satisfy the above requirement. To overcome this problem, the indexed space is extended to 256×256. The extra "padding" is simply ignored because all queries lie within the 240×240 area. The Hilbert curve for Weathr has been implement using an



(a) Hilbert curve indexing an 8×8 grid.



(b) Hilbert curve extract of Figure 12(a) showing three subcurves: 8–11, 30–32, and 53–54.

Figure 12: Hilbert space-filling curve.

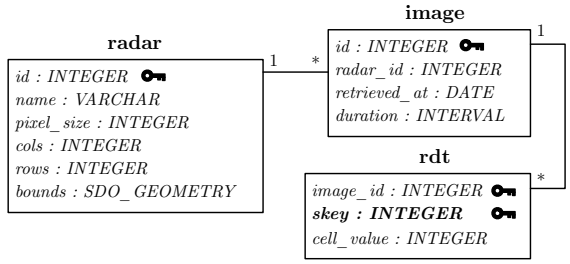


Figure 13: Hilbert-based schema. The only real change from Figure 9 is the *skey* attribute, marked in bold font.

efficient non-recursive algorithm that utilizes a for loop and bitwise operations [26, 27].

The Hilbert-based queries operate upon data stored in tables with a schema illustrated in Figure 13. Note that, compared to Figure 9, the only change is that the column *skey* replaces the columns x and y. When performing queries it is therefore necessary to map coordinates in the two-dimensional 240×240 grid to one-dimensional *skey* values. Put another way, a pixel (x, y, v) in the LAWR raster image is queried by mapping the coordinate (x, y) to an *skey* value and using this as a search key to perform the lookup. The mapping is depicted in Figure 12(b) where, e.g., the coordinate $(2, 2)$ is mapped to $skey = 8$. A Java stored function located on the database server is responsible for performing the mapping. The function has input values x, y, and l, where l is the so-called *level* which determines the size of the indexed space—e.g., level 8 indexes a space with side-length $2^8 = 256$.

On query-time a set of *skey*s can be grouped into segments that consist of numerically adjacent *skey*s and thereby avoid having to retrieve each precipitation value one at a time. This is illustrated in Figure 12. The gray area in Figure 12(a) represents a spatial range query. Figure 12(b) shows a close-up of the same range query. The query's spatial range covers nine pixels that each have their own unique *skey* value. The nine keys can be grouped into three unique segments consisting of numerically adjacent *skey* values: 8–11, 30–32, and 53–54. Due to the locality-preserving properties of the Hilbert curve, a typical user-created polygon will be made of fewer segments than it contains pixels. If data is organized properly on disk, this will result in fewer random block reads when fetching a spatial region.

4.3.1 Indexing

The choice of indexing scheme for the Hilbert-based solution is identical to that of the coordinate-based solution described in Section 4.2.1. However, where the composite key of the **rdt** table in the coordinate-based solution is a triple, the composite key of the Hilbert-based solution is a pair consisting of the foreign key column *image_id* and the *skey* column.

Since the two-dimensional coordinates of a pixel are now expressed using a single integer, the storage of each pixel requires less disk space, making the space savings even more verbose by applying the IOT concept on the Hilbert-based solution. In fact, when comparing the allocation sizes of the primary key index of the non-IOT solution and the

IOT solution, identical size were observed despite all data being stored in the index segment in case of the IOT.

On IOTs *rowid*s are not present. They do not make sense since data is stored inside the index itself. Instead the primary key takes its place and serves as a logical *rowid*. In our case, the size of the primary key is less than that of a *rowid*, which effectively means that data is physically organized on disk in primary key order while saving disk space at the same time. Range queries are especially believed to gain from applying an IOT due to the locality-preserving features of the Hilbert space-filling curve and data being organized on disk in search key order.

4.3.2 Range Query

The Hilbert-based spatiotemporal range query implementation strongly resembles that of the range query in Section 4.2.2. However, the Hilbert-based algorithm differs since it does not establish a scan line order on the pixels positions output by the rasterization algorithm. The algorithm instead maps each pixel position from the rasterization output to an *skey* value and then groups these into segments of Hilbert subcurves.

Listing 3 shows the function for executing a spatiotemporal range query. Lines 3–6 calculate an *skey* value for each pixel position output by the rasterization algorithm and adds them to the `skey` array. This is done using the `xy2skey` function that takes in a two-dimensional raster space coordinate and an integer defining the level, and outputs the corresponding *skey* value. Line 7 calls a function `toSegments` that splits an array of *skey* values into subcurves, as illustrated in Figure 12(b). Once the `toSegments` function has been called, the first thing that happens is that the `skey` array is sorted in ascending order, preparing the array for a loop structure in lines 16–21 that finds the subcurves in the `skey` array. Line 17 checks if an *skey* and its predecessor in the `skey` array are numerically adjacent. If this is the case, the two *skey* values lie on the same subcurve, and the algorithm simply continues. If not, the two *skey* values lie on different subcurves, and the appropriate segment is added to the `segment` list in line 18. After exiting the loop, the final segment is added in line 22. After this, the `toSegments` function terminates and returns control to the `hilbertRange` function. In line 8 an SQL string equivalent to the relational algebra statements in Query 3 is built, where b is the smallest *skey* value of a segment in s, and e is the largest. Finally, the result of the SQL query is returned in line 9.

$$
\begin{aligned}
\text{join} &\leftarrow image \bowtie_{image.id=rdt.image\_id} rdt \\
\text{temp}_0 &\leftarrow \sigma_{t \geq t_1 \wedge t \leq t_2}(\text{join}) \\
\text{temp}_1 &\leftarrow (\sigma_{s \geq b_0 \wedge s \leq e_0}(\text{temp}_0)) \cup \\
&\quad\ (\sigma_{s \geq b_1 \wedge s \leq e_1}(\text{temp}_0)) \cup \\
&\quad\ \vdots \\
&\quad\ (\sigma_{s \geq b_n \wedge s \leq e_n}(\text{temp}_0)) \\
\text{result} &\leftarrow \Pi_{t, t\mathcal{G}_{\mathbf{sum(v)}}}(\text{temp}_1)
\end{aligned}
$$

Query 3: Relational algebra statements for performing a Hilbert-based range query.

```
1  hilbertRange(geom:SDO_GEOMETRY; t1,t2:
       TIME_TYPE; l:INTEGER):SQL_RESULT is
2   do
3    pixels:=rasterize(geom);
4    for i:=0 to pixels.length-1 do
5     skeys[i]:=xy2skey(pixels[i],l);
6    end;
7    segs:=toSegments(skeys);
8    sql:=buildQueryString(segs,t1,t2);
9    return dbExecute(sql);
10 end;
11
12 toSegments(skeys:ARRAY):LIST is
13  do
14   sort_asc(skeys);
15   segStart:=skeys[0];
16   for i:=1 to skeys.length-1 do
17    if (skeys[i-1]≠(skeys[i]-1))
18     segs.add(segStart,skeys[i-1]);
19     segStart:=skeys[i];
20    end;
21   end;
22   segs.add(segStart,skeys[skeys.length-1]);
23   return segs;
24  end;
```

Listing 3: Hilbert-based range query.

4.3.3 Single-Point Query

The implementation of the Hilbert-based single-point query differs only very slightly from that of the coordinate-based single-point query. The only difference is the addition of a call to the `xy2skey` function that converts a two-dimensional raster space coordinate to an *skey* value. This *skey* value is then used for performing a lookup in the database. The actual query executed is equivalent to the relational algebra statements in Query 4, where t_1 and t_2 define the temporal range and s_0 is the *skey* value output by the `xy2skey` function.

$$
\begin{aligned}
\text{join} &\leftarrow image \bowtie_{image.id=rdt.image\_id} rdt \\
\text{temp}_0 &\leftarrow \sigma_{t \geq t_1 \wedge t \leq t_2}(\text{join}) \\
\text{temp}_1 &\leftarrow \sigma_{s=s_0}(\text{temp}_0) \\
\text{result} &\leftarrow \Pi_{t,v}(\text{temp}_1)
\end{aligned}
$$

Query 4: Relational algebra statements for performing a Hilbert-based single-point query.

```
1  hilbertkNN(origin:COORD_TYPE; t1,t2:
       TIME_TYPE; k:INTEGER):SQL_RESULT is
2   do
3    running:=true;
4    kFound:=0;
5    ringlevel:=0;
6    skeyOrigin:=xy2skey(origin);
7    while (running) do
8     ring:=getRingSkeys(ringLevel,skeyOrigin);
9     sql:=buildQueryString(ring,t1,t2);
10    tmpSkeys:=dbExecute(sql);
11    for (i:=0 to tmpSkeys.size()-1) do
12     subResult.add(skey2xy(tmpSkeys.get(i)));
13    end;
14    kFound:=kFound+subResult.size();
15    kNNResult.union(subResult);
16    if(kFound ≥ k)
17     running:=false;
18    end;
19    ringLevel++;
20   end;
21   return kNNResult;
22  end;
```

Listing 4: Hilbert-based k-NN query.

4.3.4 k-NN Query

Just like the previous query types, a Hilbert-based k-NN query is similar to its coordinate-based counterpart. But since it is based on a space-filling curve, it relies on the `xy2skey` function. However, it also makes use of the reversed `skey2xy` function since a k-NN query is location-aware and must return the positions of its k or more results, requiring a calculation $\mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$.

Listing 4 shows the pseudo code for performing a Hilbert-based k-NN query. The basic principle of the algorithm is the same as for a coordinate-based k-NN query, but there are some extra steps which are explained in the following. The calculation of the *skey* values covered by a ring is handled by the function `getRingSkeys` in line 8. A ring in this context is exactly the same as in the coordinate-based k-NN query—see Figure 11—only instead of returning pixel positions, the function `getRingSkeys` returns the corresponding *skey* values. Lines 9–10 build and execute an SQL statement almost exactly like that of the Hilbert-based range query—see Query 3. The only difference is a slightly modified projection at the end:

$$\text{result} \leftarrow \Pi_{s,s\mathcal{G}\mathbf{sum(v)}}(\text{temp}_n)$$

The result of the query is a row per pixel (s, v_{sum}), where v_{sum} is the sum of each individual pixel within the specified temporal range. In lines 11–13 each row (s, v_{sum}) is converted to a row (x, y, v_{sum}) using a function `skey2xy`. The rest of the algorithm is completely identical to the latter part of the coordinate-based k-NN query algorithm.

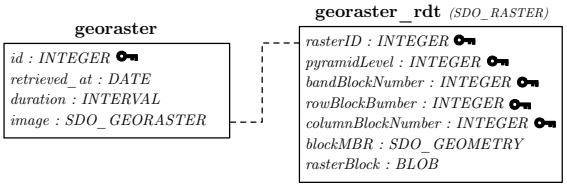


Figure 14: Schema for the GeoRaster based solution.

4.4 GeoRaster-Based Solution

GeoRaster [2] is a component of Oracle Spatial [28] that enables the user to store, index, query, analyze, and deliver raster images and other gridded data along with associated metadata. Raster data can be stored by GeoRaster in a number of different image formats and georeferenced to positions on the Earth's surface or to a local coordinate system. GeoRaster is typically used to store images from technologies that capture or generate raster images, e.g., remote sensing.

For our purposes, GeoRaster acts as a COTS solution that facilitates an implementation of database integration in the Weathr application. This implementation is a valuable reference, providing a performance baseline against which to compare the coordinate- and Hilbert-based solutions. Using GeoRaster, raster LAWR images are georeferenced to a position on the Earth's surface. This allows a query to, e.g., return a LAWR pixel value when given a location on the Earth's surface associated with that pixel. Historical queries can be answered through GeoRaster by continuously storing incoming LAWR images in the system, and then subsequently querying the relevant data using the GeoRaster Java API [29].

Physically, the GeoRaster data model consists of two data types and an object-relational schema. For each raster image, a row is stored in a user-defined table containing a column of object type

SDO_GEORASTER along with any number of user-defined columns. In the schema of Figure 14, this table is the **georaster** table. The column *image* is of type SDO_GEORASTER and contains information about the raster image. The columns *retrieved_at* and *duration* are user-defined columns equivalent to the columns of the same name on the image table of the coordinate- and Hilbert-based solutions. Compared to the latter two solutions, there are no columns defining image properties such as the number of columns, rows, and pixel size. These are all enclosed in the SDO_GEORASTER object. Furthermore, spatial reference information is also enclosed inside this object, which is why a *bounds* column of type SDO_GEOMETRY is not included.

Each SDO_GEORASTER object includes a number of SDO_RASTER row type objects, each of which represent a single raster *block*. Where the two previous solutions saves a pixel per row in the RDT, GeoRaster stores the images in chunks in a binary large object whose dimensions are defined by a blocking size option. As documented by [30], GeoRaster will generally perform better when the blocking size is close to the typical query size. In our case, the blocking size has been set to 16×16. Each block has a range of attributes attached—*rowBlockNumber* and *columnBlockNumber* are equivalent to the x and y columns of the coordinate-based solution, only they describe the position of a block and not a pixel. The attributes *pyramidLevel* and *bandBlockNumber* are not used in the context of Weathr. Pyramids is a feature of GeoRaster for building pyramid structures from the images that it stores, delivering different resolutions depending on, e.g., zoom level in a GIS application. Furthermore, the images received from a LAWR radar are single-banded, meaning that the *bandBlockNumber* attribute will remain zero throughout. Each SDO_RASTER object is saved in an RDT named **georaster_rdt** in Figure 14. No relational constraints are depicted in the figure since none exist in the GeoRaster object-relational schema. Instead, the RDT's integrity is maintained by DML triggers on the *image* column of the **georaster** table.

GeoRaster offers different ways of querying raster image data stored in the database, of which the three most relevant ones are: (i) A procedure SDO_GEOR.getRasterData returns through an OUT parameter an entire raster image as a BLOB object. Using this procedure in our scenario would basically reduce GeoRaster to an advanced BLOB storage mechanism. (ii) A function SDO_GEOR.getCellValue returns the value of a single pixel in the raster image when supplied with either a raster space coordinate—i.e., $(\text{row}, \text{column})$—or a point geometry of type SDO_GEOMETRY. Preliminary tests have shown that this function has a time complexity of $O(n)$, where n is the amount of pixels in an image, meaning that querying subsets of an image is very slow when using the getCellValue function. In other words, querying 400 pixels takes about 1.5 seconds, and querying an entire 240×240 image takes over 3 minutes. This is obviously not acceptable for querying large portions of an image. Finally, (iii) a procedure SDO_GEOR.getRasterSubset returns through an OUT parameter a subset of a raster image by passing a *window* parameter from which to crop the pixels. A window is defined by upper-left and lower-right coordinates as specified by raster space coordinates or the MBR of a geometry object of type SDO_GEOMETRY. Also, a window can be defined using positions on the Earth's surface if valid Spatial Reference Identifier (SRID) values are defined for the geometry object. The getRasterSubset procedure returns an image encoded using a variety of well-known graphics file formats (TIFF, GeoTIFF, PNG, etc.), and output is limited to the rectangular nature of the MBRs surrounding geometry objects. This means that, when using GeoRaster, post-processing must be performed in order to filter out excess pixels, since we can not specify an arbitrarily-shaped window geometry [31]—see Figure 15.

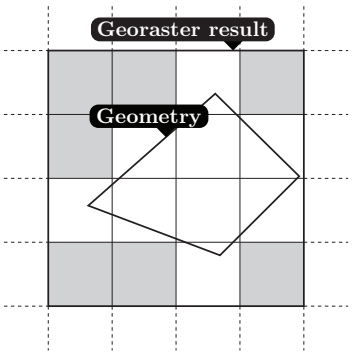


Figure 15: Post-processing when using GeoRaster. Excess pixels (colored gray) are filtered out from the result set.

4.4.1 Indexing

The indexing schemes applied on top of the GeoRaster solution are recommendations from Section 3.7 of the GeoRaster Developer's Guide [31]. It is recommended that each RDT is applied a B-tree index on the columns *rasterId*, *pyramidLevel*, *bandBlockNumber*, *rowBlockNumber*, and *column-*

```
1 GeoRasterRange(geom:SDO_GEOMETRY; t1,t2:
      TIME_TYPE):SQL_RESULT is
2  do
3   geoResult:=queryMBRData(geom,t1,t2);
4   pixels:=rasterize(geom);
5   for i:=0 to geoResult.size()-1 do
6    result.union(filter(pixels,geoResult.
         getRow(i)));
7   end;
8   return result;
9  end;
```

Listing 5: GeoRaster-based range query.

```
1 GeoRasterkNN(center:COORD_TYPE; t1,t2:
      TIME_TYPE):SQL_RESULT is
2  do
3   running:=true;
4   kFound:=0;
5   ringLevel:=0;
6   while (running) do
7    ring:=getRing(ringLevel,center);
8    subResult:=queryRingData(ring,t1,t2);
9    kNNResult.union(subResult);
10   kFound:=kFound+subResult.size();
11   if (kFound ≥ k)
12    running:=false;
13   end;
14   ringLevel++;
15  end;
16 end;
```

Listing 6: GeoRaster-based k-NN query.

BlockNumber. This indexing is obtained implicitly since these columns have been defined as primary keys of the RDT. Furthermore, a spatial index on the *spatialExtent* attribute of the `SDO_GEORASTER` object should be applied. The latter is highlighted as the most important index. In the scenario of WEATHR, however, we do not expect any particular gains from this latter index during performance testing. This is due to all LAWR images currently being georeferenced in exactly the same position, since there is only one LAWR, rendering the spatial index inconsequential.

4.4.2 Range Query

A spatiotemporal range query in the GeoRaster-based solution basically consists of extracting a subset of a number of images by supplying a window parameter, and subsequently filtering out excess pixels. First, a geometry and a temporal range is supplied by the user. Then a rectangular subset of raster data is returned by the `getRasterSubset` procedure by passing it the MBR of the geometry as a window parameter. Second, the geometry is rasterized, identifying exactly which pixel values need to be part of the query result. Finally, the GeoRaster result set is iterated through, filtering out pixels that do not overlap the rasterization from the previous step. The result is all pixel values that overlap the original geometry. This process is repeated for all images in the provided temporal range.

Listing 5 shows the algorithm for executing a GeoRaster-based range query. The algorithm is relatively abstract, concealing most of the logic behind a call to the `queryMBRData` function in line 3. In short, this function is responsible for querying the database for raster data using the GeoRaster SQL API. In order for it to do this, it is passed three arguments: a user-created geometry defining the spatial range and two timestamps t_1 and t_2 defining the temporal range. For each image within the temporal range, a subset of raster data is extracted by calling the GeoRaster `getRasterSubset` procedure mentioned earlier, passing it the MBR of the geometry as a window parameter. In line 4 the input geometry is rasterized. The result of the rasterization is used in conjunction with the extracted raster data in lines 5–7, where excess pixels are filtered out of the raster data by the `filter` function, as illustrated in Figure 15. Finally, in line 8, the result of the range query is returned.

4.4.3 Single-Point Query

A single-point query using GeoRaster is implemented using the function `getCellValue` mentioned earlier. The rasterizer component takes in a real-world coordinate and outputs the corresponding pixel position. This pixel position is then fed to the `getCellValue` function, which in turn outputs the appropriate pixel value.

4.4.4 k-NN Query

The GeoRaster-based k-NN query is similar to the coordinate-based k-NN query. It is based on the same principle of fetching rings of data, one ring at a time. The only real difference lies in the way data is queried. Using GeoRaster, data is queried by repeatedly calling the `getCellValue` function. An implementation that instead utilizes the `getRasterSubset` procedure is also feasible, but that would have implied that each fetch of a ring of data consist of also fetching everything inside the ring.

The algorithm for executing a GeoRaster-based k-NN query is shown in Listing 6. It is virtually identical to Listing 2, except for line 8 where data is queried by the `queryRingData` function.

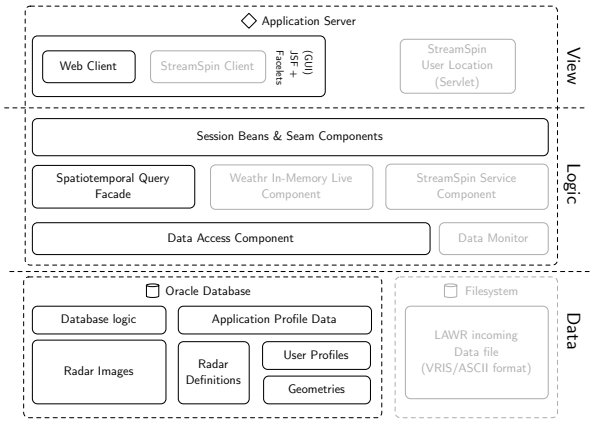


Figure 16: Overall system architecture.

Querying data in this case consists of calling the `getCellValue` function on the pixels covered by the ring calculated in line 7. Note that this is done for all images within the temporal range defined by t_1 and t_2. Additionally, pixels returning a value of zero are filtered out from the result of the `queryRingData` function. The rest of the algorithm proceeds as previously explained.

5 Architecture and Integration

The previous section dealt with the design of algorithms and database schemas. This section broadens the scope and deals with the implementation and integration into the prototype introduced in [1].

5.1 Application Architecture

The base of the application architecture is drawn from [1], presenting a highly modular design based on a three-tier architecture, separating concerns into view, business-logic, and persistence. Figure 16 shows an overview of the current prototype architecture. For a detailed description of the overall system architecture—in particular the grayed out components—consult [1]. Where the usual three-tier architecture describes a clear separation of concerns, this prototype breaks the convention somewhat since portions of the domain specific logic has been moved into the DBMS. This is, more specifically, the logic concerned with spatiotemporal querying as presented in Section 4.

The Weathr application is built on the JBoss Seam application framework, which is a framework aimed at developing rich internet applications that integrates a wide variety of components and standards from the Java community into a complete enterprise application framework. The front-end is implemented by utilization of JSF/Facelets, which is a component-based view technology.

The model of the application is tightly integrated with the database through the Hibernate Object Relation Mapping (ORM) tool, which serves as part of the JBoss implementation of the EJB3 specification. Hibernate conforms to the JPA (Java Persistence API), but delivers some additional extensions which come in handy for implementing the data access layer.

5.2 Data Layer

Moving from the bottom of Figure 16, we start out by presenting the Oracle Database layer.

5.2.1 Oracle Database

The table design of the prototype is in close correspondence to that of the ER diagram and schema designs presented in Section 4. In addition, a few other tables are present. Geometries are constructed from the coordinates delivered from Google Maps. These geometries are represented in latitude/longitude coordinates in a Google-specific mercator projection. Internally the components operate in the Universal Transverse Mercator (UTM) projection. Since coordinate reference system conversion is irreversible without introducing precision loss, the original geometries are stored in the **geometry** table. A database view is not applicable for handling this, since a spatial index cannot be applied on views and is needed for a spatial join process between radar bounds and the geometries used as a prefiltration step before each query. Materialized views do not allow update on commit time when a function is involved. Instead, a trigger managed table is kept of the converted geometries in UTM projection. Furthermore, temporary tables are used for intermediate storage for k-NN queries and for storage of geometries for non-saved queries.

The query logic for all three solutions is implemented primarily by the use of Java stored procedures in correspondence to the pseudo code of Section 4, returning either result cursors or SQL table types.

5.3 Application Server

Stepping up in Figure 16, we have the prototype itself which runs on a JBoss application server. In the following section, the data access component

integrating the link between the database and the application is described.

5.3.1 Data Access Component

The Data Access component can be separated into three parts: (i) An entity model, (ii) Data Access Objects (DAO), and (iii) the Object Relational Mapping (ORM) engine.

All communication between the application and the database takes place through the data access component which comprises a set of annotated Hibernate entity objects modeled according to the presented ER diagram in Figure 8 as to reflect the database schema design.

All geometries in the model are mapped through Hibernate ORM with the extension project named Hibernate Spatial [32], which is a set of tools providing mapping between vendor-specific spatial systems such as Oracle Spatial, PostGIS, and the Java Topology Suite (JTS) [33], which is a light, yet powerful, spatial library for Java that conforms to the "Simple Features Specification for SQL" by the Open GIS Consortium [34]. This adds the possibility of using the JTS geometries as attributes on the entity objects and have Hibernate persist them to disk—in this case as the Oracle-specific `SDO_GEOMETRY` spatial type.

As previously mentioned, the spatiotemporal queries are performed on the database through a number of stored Java functions. Normally, entity objects map to corresponding tables on the database. In order to execute the stored Java functions, one possibility would have been to retrieve a connection through the Hibernate entity manager and issue direct queries through JDBC and subsequently unmarshalling the data to objects. However, Hibernate supports mapping native queries and callable statements to entity objects as long as a database cursor is returned at the first parameter index of the statement. This is simply implemented as an entity bean for the range query and another for the k-NN query, each with the proper Hibernate annotations and attributes corresponding to the return values for the particular query type.

All DAOs inherit from an abstract generic class implementing the most basic queries such as selection of single entities based on the primary key identifier, persisting entities, and loading all entities.

Present are DAOs for, e.g., retrieving users based on different search criteria and a trivial one for loading user roles. The most interesting DAO, however, is the `ImageDAO` which, besides retrieving the image data into the application, contains the code for executing the spatiotemporal queries. These access methods are listed in Listing 7. Line 1 carries out a range query given a geometry ID reference to a stored geometry. Line 2 is an overload of the previous method, carrying out the query on a non-saved geometry. Line 3 issues a k-NN query given an id reference to a point geometry ID, a date range, a value k, and lower and upper bound for the precipitation levels that must be considered results.

```
1 List<RangeQueryObject> range(int geomId, Date
      from, Date to);
2 List<RangeQueryObject> range(Geometry geom,
      Date from, Date to);
3 List<KNNObject> knn(int pointId, Date from,
      Date to, int kResults, int lower, int
      upper);
4 List<KNNObject> knn(Point point, Date from,
      Date to, int kResults, int lower, int
      upper);
```

Listing 7: Spatiotemporal DAO Interface.

5.3.2 Spatiotemporal Query Facade

The spatiotemporal query facade is a quite shallow component, since all query logic is contained in the database. The interface is equal to that of the `ImageDAO`. It merely exists for the purpose of complying with the architecture of the rest of the prototype and for accommodating future extensions. It is basically a single class which is the entrance point for all other parts of the application.

5.3.3 Session Beans and Seam Components

The session beans and seam components depicted in Figure 16 forms the entrance point for the view layer components. These components reflect the different tasks from a user's point of view, such as user registration and authentication, administration, and most importantly of all, a component for issuing queries both on live and historical data.

5.4 User Interface

The user interface has been extended from that presented in [1], adding animated forecasts, a push-style update of data on the interface when new data arrives, utilizing Java Messaging features of the Seam Remoting component. Most important of all is that users are now able to issue historical queries on both saved and non-saved geometries. Whenever the user issues, e.g., a historical range query, he chooses an already created geometry or creates one by clicking the Google Maps interface. He then specifies a temporal range and hits a submit button. The result of the range query is re-

turned as a chart showing the precipitation levels within the geometry throughout the provided temporal range.

6 Performance Study

This section describes the results from a number of tests carried out in order to study the performance of the three solutions devised for performing historical queries in WEATHR. Performance is measured on several factors. In the first test, the space usage of each individual solution is measured. In the second, the query execution performance of spatiotemporal range, single-point, and k-NN queries is tested. First, however, a brief summary of the test environment is presented.

6.1 Test Environment

All tests have been carried out on a machine equipped with a quad core Intel Xeon 2.13 GHz CPU, four hard disk drives, and 4 GB memory. The machine is running the Debian GNU/Linux operating system, the JBOSS Application Server, and Oracle Database 11G.

The WEATHR database instance is configured to use automatic shared memory management and is equipped with a shared memory space roughly half the size of the amount of physical memory (∼2 GB). Furthermore, database log settings have been changed from their defaults, storing the logs on a separate disk in order to minimize disk I/O impact when maintaining log files.

Finally, in order for the cost-based optimizer (CBO) to make the best execution plan for our SQL queries, statistics are gathered from the tables and indexes that participate in the queries. This is done by calling `dbms_stats.gather_schema_stats` just before executing the tests. Note that no additional data arrives at the system while the tests are running, ensuring that the gathered statistics are always up to date and thereby enabling the CBO to consistently make an informed decision.

6.2 Space Usage

The purpose of this first test is to outline the space requirements of each solution. Given the append-only storage requirement of the prototype—i.e., new LAWR data sets constantly being stored in the database, never to be deleted—a space efficient storage mechanism is of great value. The coordinate- and Hilbert-based solutions implement almost identical database schema layouts, both storing a single pixel per row, whereas the GeoRaster-based solution differs by storing images in blocks, where each block is a subset of the original image. Furthermore, due to the way GeoRaster stores images—i.e., using image compression techniques such as JPEG or Deflate—all pixel values from the original images are stored, including zero-values. A test conducted in [1] shows that approximately 97.2 percent of the values in 18 days worth of data (a total of 5,184 data sets) were zero-values, meaning that the coordinate- and Hilbert-based solutions only store approximately 2.8 percent of the entire amount of LAWR data.

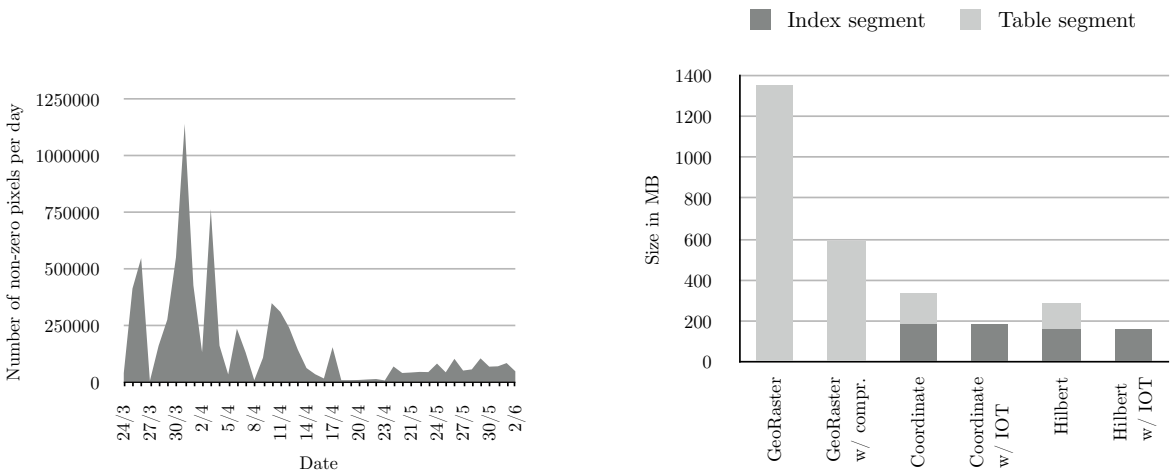
Figure 17(a) shows the distribution of the data used for measuring space requirements. The data spans a period from March 24th to June 2nd, during which a LAWR data set has been received once every five minutes. Note that due to LAWR downtime during that period, the actual number of data sets received is 12070. Figure 17(b) contains a graph showing the space requirements for the RDT of all three solutions. The four rightmost bars represent the Coordinate- and Hilbert-based solutions—two bars for each, showing space usage using a heap table and an IOT, respectively. As expected when using a heap table, the index segment takes up roughly as much space as the actual data stored in the table segment. In terms of storage requirements, it is clear that IOTs represent a massive reduction, requiring approximately half that of a heap table. Furthermore, the Hilbert-based solution has a slightly lower space usage due to the representation of a two-dimensional coordinate using a one-dimensional *skey* value. The two leftmost bars show the space usage of the GeoRaster RDTs. Both use considerably more space compared to the other RDTs, especially when storing uncompressed data. But even when compressing the data using the Deflate compression algorithm, the GeoRaster RDT uses almost four times the space compared to the Hilbert RDT using an IOT.

6.3 Querying

In this section, the three query types—i.e., range, single-point, and k-NN—are tested for execution time. The LAWR image data used for testing consists of actual recorded data as to assure the tests do not fall far from a real-world scenario.

6.3.1 Range Query

Performance testing the spatiotemporal range query consists of two subtests. The first subtest is designed to emulate a real-world scenario during heavy load. It has been performed by concurrently querying 500 predefined geometries with sizes and



(a) Data statistics.

(b) Space usage per solution.

Figure 17: Space usage.

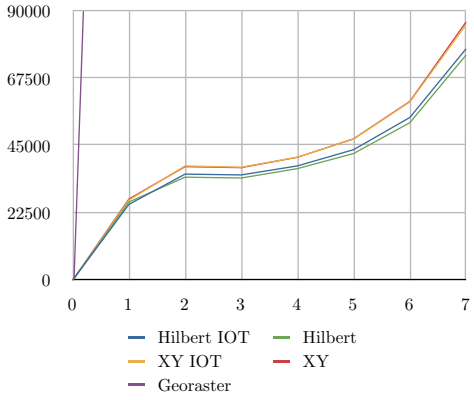


Figure 18: 500 concurrent range queries.

shapes that can be described as being "realistic"—covering fields, small towns, buildings etc., having an average pixel coverage of $\sim 5\frac{3}{4}$. The geometries are distributed across the entire range of the LAWR, with only few overlaps. The test is repeated with temporal ranges that range from a single day to seven days in single-day steps—ultimately covering March 25th to April 1st.

In case of the coordinate- and Hilbert-based solutions, the range query is tested using both regular heap tables as well as IOTs. It is expected that the IOTs perform slightly faster than heap tables due to them being organized on disk in search key order, making a query consist of less random block reads due to the index and table segments being merged together. The GeoRaster-based solution does not have such advantages when executing a spatiotemporal range query, and must in addition perform a filter step to eliminate excess pixels.

Figure 18 presents the results from the first subtest. The GeoRaster graph is completely off the chart, consuming a whopping 1046101 ms ($\sim 17,5$ minutes) when executing 500 concurrent range queries with a single-day temporal range. The upward curvature on the remaining graphs is presumably due to an increasing data amount around March 31st to April 1st—see Figure 17(a). There are no noticeable differences in the performance graphs of the IOT and heap table graphs. Some tests have shown results in favor of IOT and some the opposite. The explanation is that data has been loaded into the heap tables in sorted order as to reflect the append only nature of the system, mimicking the organization of the IOT. This is, however, a pleasing result considering the fact that an IOT consumes roughly half the space of a regular heap table with an index on top. But perhaps more interesting is the fact that the performance of the Hilbert-based solution compares to that of the coordinate-based solution, meaning that using a Hilbert space-filling curve for indexing spatial raster data has seemingly no worthwhile effect. A likely reason for this lies with the size of "realistic" geometries. With a pixel size of 500m $\times$ 500m, such a geometry covers only few pixels and does therefore not gain from constructing SQL statements from segments of *skey*s. Figure 19 shows the distribution of the various segment sizes found using the scan line algorithm of the coordinate-based solution and the subcurve algorithm of the Hilbert-based solution, respectively. The Hilbert-based solution produces the most segments of size one, meaning that it also requires the most random block reads.

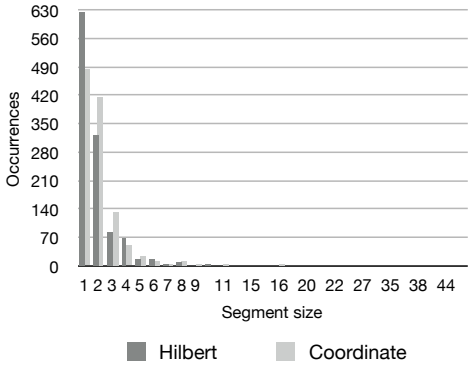


Figure 19: Segment size distribution in 500 geometries.

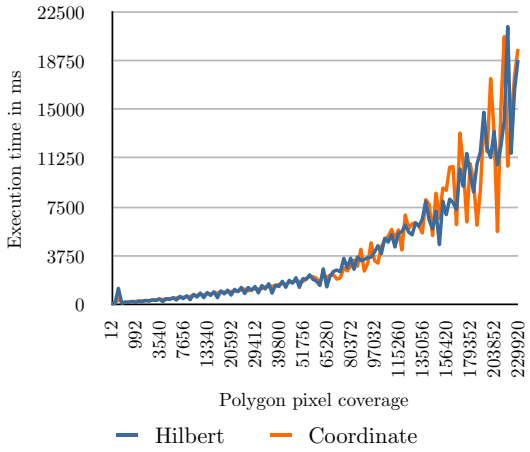


Figure 21: Varying geometry pixel coverage.

An interesting side note is that reversing the composite index—e.g., (*skey*, *image_id*)—would provide a performance benefit, causing the entire temporal range for a single pixel to lie adjacently in the index. Results from an identical test with a reverse composite index applied are presented in Figure 20. The reversed index is clearly faster, yielding nearly constant performance as the temporal range increases. This is due to the entire temporal range for each pixel being read using a single sequential block read. Furthermore, during execution of the first subtest, a substantial amount is spent on CPU activity with only little I/O, meaning that the spatiotemporal range query is CPU bound. This suggests that the `GROUP BY` statement and the `SUM` aggregate function are the most expensive operations in the SQL statement.

The goal of the second subtest is to further measure the consequence of using a space-filling curve for indexing two-dimensional raster space. The locality-preserving behavior of the Hilbert curve ensures that data is indexed in such a way that the

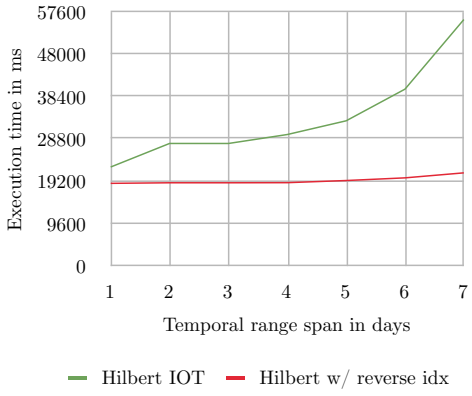


Figure 20: 500 concurrent range queries—with and without reverse index applied.

pixels covered by a geometry will be located in close proximity in the index, as described in Section 4.3. Ultimately this should lead to faster performance when querying a spatial region. It is expected that as geometries grow larger, the Hilbert curve will show increasingly larger performance benefits. This expectation is based on the fact that the larger a geometry becomes, the more pixels it will cover, and in return the larger the subcurves of adjacent *skey*s will become. Additionally, considering that the Hilbert curve moves in either a horizontal or vertical direction, the largest subcurves are found in rectangular geometries. For the purpose of this subtest, data sets with larger resolution are generated by recursively splitting the pixels of the original 240×240 LAWR data sets, giving an RDT with pixel size 125m $\times$ 125m and dimensions 960×960. The desired effect of this is to make geometries cover a larger amount of pixels due to the smaller pixel size. 120 rectangular geometries with increasing pixel coverage are generated and queried with a temporal range used covering a single image only, ensuring a focus on spatial query performance.

Figure 21 presents the results from the second subtest. As can be seen from the two graphs, the Hilbert- and coordinate-based solutions perform very similarly, apart from some fluctuations. This is unfortunate in case of the Hilbert-based solution, since the test results clearly show that the use of a Hilbert curve for indexing two-dimensional raster space does not yield the desired performance benefit. It seems that the scan line order in which pixel values are read in the coordinate-based solution is every bit as efficient as utilizing the locality-preserving behavior of the Hilbert curve. This brings up a discussion on the characteristic of the way LAWR data is stored in the database. Because

	Coordinate	Hilbert	GeoRaster
1 day	121 ms	138 ms	3646 ms
3 days	126 ms	131 ms	10568 ms
5 days	128 ms	130 ms	17415 ms
7 days	146 ms	143 ms	22413 ms

Table 1: Single-point query with increasing temporal range.

only non-zero pixel values are present in a RDT and non-zero values are few and far between in a typical LAWR image, the data is best described as being very sparse. As a possible explanation of the outcome of the test, consider that even though a geometry consists of large Hilbert subcurves, many of these curves will cover only few rows in the RDT, resulting in far smaller sequential block reads. Had the data been dense, a large subcurve would result in a sequential read fetching all rows covered by the subcurve, allowing the locality-preserving behavior of the Hilbert curve to improve performance.

6.3.2 Single-Point Query

The test devised for the single-point query type is quite simplistic. No substantial difference in performance is expected between the coordinate- and Hilbert-based solutions, since the implementation of each is very similar. Furthermore, a single-point query is unable to take advantage of any locality-preserving indexing, since only a single pixel is queried for each LAWR data set in the temporal range. The GeoRaster-based solution, on the other hand, is expected to perform several magnitudes slower in comparison. GeoRaster uses the `getCellValue` function, which preliminary tests have shown to be a slow performer—see Section 4.4.

Table 1 shows the results from performing a number of single-point queries with increasing temporal range. The outcome is as expected. GeoRaster is slowest, taking up to over 150 times as long to complete compared to the coordinate- and Hilbert-based solutions. The two latter have comparable performance.

Just as with the range query, the single-point query is expected to benefit from using a reversed composite index. Such a test has been performed, but the result did not reveal any discerning difference compared to using a non-reversed composite index. The reason for this is most likely caused by only a minuscule amount of data being queried, concealing any performance differences.

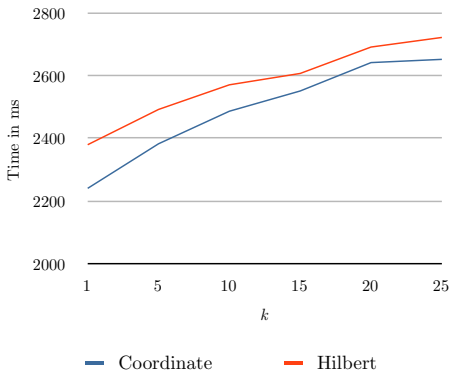
6.3.3 k-NN Query

As mentioned in Section 4.2.4, a k-NN query relies on querying increasingly larger rings of data, until a minimum of k results have been found or the ring exceeds the boundaries of the data set. However, the coordinate-based solution offers an optimization because pixels are arranged in a scan line order, allowing a sequential read at the top and button of each ring, and thereby reducing the number of random block reads.

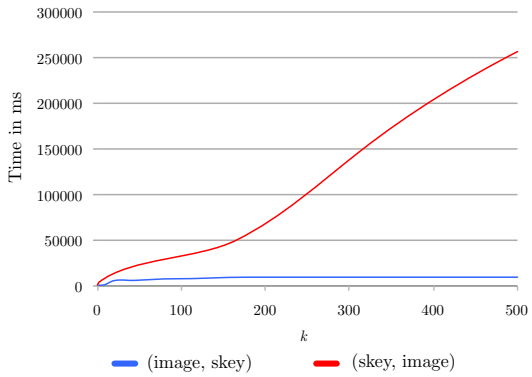
Figure 11 illustrates how rings are built. L_3 is the third ring to be queried and consists of 24 pixels. By performing sequential reads at the top and button rows of pixels in the ring, more than half the pixels in the ring are fetched in two read operations. This repeats itself in all rings visited by a particular query. The remaining pixels in a ring are, however, fetched by random block reads. As a purely speculative note, consider the remaining pixels in a relatively small ring. Due to the closeness of the pixels, these pixels may actually be fetched in a sequential read if the database optimizer deems this more cost-effective than performing additional index lookups.

The locality-preserving behavior of the Hilbert curve is not likely to provide measurable cost savings on k-NN queries since a ring of data does not follow the sequential ordering of the Hilbert curve. The Hilbert subcurves covered by a ring will typically have a length of just one or two pixels, whereas only very few have a length of three pixels or more. Turning to the GeoRaster-based solution, subpar performance is once again expected since it—just as the single-point query type—relies on the slow `getCellValue` function.

Figure 22(a) illustrates the performance of the Hilbert- and coordinate-based solutions using IOTs for increasing values of k on real-world data sets. The temporal extent remains constant throughout and covers April 2nd 08:06 to April 2nd 12:06, containing a total of 48 images. The graphs show that the Hilbert-based solution is slightly slower compared to the coordinate-based solution. Within the specified temporal range there are 478 possible results to be found before reaching the LAWR images boundaries. When setting $k = 479$, both algorithms search through the entire spatial range and terminate with 478 results found, taking the coordinate-based solution 9993 ms and the Hilbert-based solution 10153 ms. During this search, 120 rings have been constructed and queried, with ring number 119 being the largest, consisting of 952 pixels. The reason for the Hilbert-based solution performing slightly slower is most likely due to performance overhead due to the conversion of *skey*



(a) Coordinate- and Hilbert-based solutions.



(b) Hilbert-based solution with different orderings on the composite index.

Figure 22: k-NN query performance.

values—a conversion that has to be performed for all the pixels in each ring query. The coordinate-based solution's advantage due to larger sequential block reads is most likely degenerated due to data being sparse. No graph is present for the GeoRaster-based solution since the resulting curve would simply not fit. At $k = 1$, the GeoRaster-based solution takes ~ 4 minutes before terminating. At $k = 5$, it takes ~ 5 minutes, and at $k = 25$ it takes $\sim 7\frac{1}{2}$ minutes.

Figure 22(b) illustrates the performance of the Hilbert-based solution when using different orderings on the composite index. The graphs show that the k-NN query exhibits superior performance when using the non-reversed composite key, i.e., $(image\_id, skey)$. The relatively short temporal range used in the test coupled with the characteristics of the expanding rings, which creates an increasingly larger spatial extent, degrades the efficiency of the reversed composite index by up to a factor of twenty compared to the non-reversed index.

7 Future Work

In order to optimize the performance of historical queries that have large temporal ranges, a proposition is to gradually reduce the temporal granularity of data by merging images as time goes. Considering that most users will probably tend to only construct high-precision queries on recent LAWR images, the temporal granularity can safely be reduced by, e.g., merging all images from an entire day into a single image when the data is more than a month old. As an added bonus, the space requirements of the prototype would as a consequence be reduced dramatically.

Another interesting direction for future work is to examine the tracking of precipitation *regions*. By intelligently dividing each LAWR image into groups of adjacent non-zero pixels, regions are formed that represent rain clouds. An R-tree variant such as the TPR-tree [35] could be used to track these regions within LAWR data sets. The TPR-tree can be used to enclose objects into MBRs with time as a parameter. A spatiotemporal query could then be answered by specifying a spatial query region q and a temporal time interval t, and thereby retrieve all objects that lie in q during t. An example of such a query is "find all rain clouds that will be in Aalborg in the next 10 minutes."

8 Conclusion

This paper has presented a prototype for performing historical queries on LAWR raster data stored in a database. The main requirement has been to find an efficient solution for querying raster data, both in terms of space usage and query execution performance.

Three functionally equivalent implementations have been made. One serves as a baseline for comparison, based on Oracle's GeoRaster component for Oracle Database. The remaining two are based on storing pixel values in a table along with corresponding two-dimensional coordinates or one-dimensional Hilbert keys, respectively. Three query types are supported: spatiotemporal range, single-point, and k-NN.

The GeoRaster-based implementation is by far the worst performer of the three, typically performing several magnitudes slower. The Hilbert-based implementation was initially expected to outperform the implementation based on two-dimensional

coordinates. Indexing two-dimensional raster space using a Hilbert space-filling curve did, however, not yield performance benefits. When querying spatial data that has been indexed using a Hilbert curve, the size and shape of the search window is essential in terms of performance. The bigger the window, the longer the individual Hilbert subcurves covered by the window will become, which leads to fewer random block reads and more sequential ones. Also, rectangular search windows will consist of longer Hilbert subcurves than concave or bulging convex ones. This led to another observation—using large rectangular search windows in the WEATHR scenario did not increase performance. A possible explanation for this is that indexing *sparse* raster data using a Hilbert curve will possibly diminish performance. In other words, when data is sparse, a typical subcurve will often cover much fewer pixels than had the data been dense, resulting in many index range scans being reduced to, in the worst case, a random block read.

Raster data in WEATHR is sparse because only non-zero values are stored, and search windows in the form of user-created geometries are often non-rectangular and most always small in size. These factors combined entail that it is undesirable to use a Hilbert space-filling curve for indexing two-dimensional raster images. In order for the Hilbert curve to show performance benefits when indexing two-dimensional raster data, the data must ideally be dense and the search window must ideally be large and rectangular.

Even though the Hilbert-based implementation did not yield the results initially hoped for, the performance of both it and the coordinate-based implementations are satisfactory, both answering queries well within an acceptable time frame.

In terms of storage requirements, the GeoRaster-based solution is the biggest offender, requiring roughly four times as much or more compared to the Hilbert-based solution using an IOT, depending on wether GeoRaster compression is applied or not.

Acknowledgment

We would like to thank Michael R. Rasmussen from the Department of Civil Engineering, Aalborg University for providing us with LAWR data sets, software for calculating extrapolated forecast data sets, and for answering questions throughout the project period.

References

[1] Dennis B. Andersen, Martin L. Kristiansen, Claus H. Poulsen, and Thomas Winterberg. Weathr: a Prototype for Location-Based Precipitation Monitoring and Warning. 2007.

[2] Qingyun Xie and Jayant Sharma. Oracle Spatial 11g GeoRaster: An Oracle Technical White Paper. 2007.

[3] Rasdaman. `http://www.rasdaman.com/`.

[4] C. Faloutsos. Gray codes for partial match and range queries. *Software Engineering, IEEE Transactions on*, 14(10):1381–1393, 1988.

[5] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 247–252, 1989.

[6] B.C. Ooi and K.L. Tan. B-trees: bearing fruits of all kinds. *Proceedings of the 13th Australasian database conference-Volume 5*, pages 13–20, 2002.

[7] Jochen Alber and Rolf Niedermeier. On multidimensional curves with hilbert property. *Theory of Computing Systems*, 33(4):295–312, 2000.

[8] D. Hilbert. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[9] D.J. Abel and D.M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Science*, 4(1):21–31, 1990.

[10] HV Jagadish. Linear clustering of objects with multiple attributes. *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 332–342, 1990.

[11] B. Moon, HV Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, pages 124–141, 2001.

[12] M.D. McCool, C. Wales, and K. Moule. Incremental and hierarchical Hilbert order edge equation polygon rasterizatione. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 65–72, 2001.

[13] S. Vassiliadis, S. Cotofana, and P. Stathis. Block based compression storage expected performance. *Proceedings of HPCS2000, Victoria*, pages 389–406, 2000.

[14] A. M. Erisman I. S. Duff and J. K. Reid. *Direct Methods for Sparse Matrice*. Oxford University Press, 1989.

[15] S. Pissanetzky. *Sparse Matrix Technology*. London-Orlando, Florida etc, 1984.

[16] G. Gundersen and T. Steihaug. Data structures in Java for matrix computations. *Concurrency and Computation Practice and Experience*, 16(8):799–815, 2004.

[17] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[18] E.G. Hoel and H. Samet. Data-parallel spatial join algorithms. *Proceedings of the 23rd International Conference on Parallel Processing*, pages 227–234, 1994.

[19] W.G. AREF and H. SAMET. Efficient Window Block Retrieval in Quadtree-Based Spatial Databases. *GeoInformatica*, 1(1):59–91, 1997.

[20] Y. Nakamura and H. Dekihara. Spatial data structures for version management of engineering drawings in CAD database. *Image Analysis and Processing, 2003. Proceedings. 12th International Conference on*, pages 219–225, 2003.

[21] StreamSpin. `http://www.streamspin.com/`.

[22] Google Maps. `http://maps.google.com/`.

[23] Oracle® Database Concepts 11g Release 1 (11.1). `http://download.oracle.com/docs/cd/B28359_01/server.111/b28318.pdf`.

[24] Shirley Ann Stern. Oracle9i Index-Organized Tables: Technical Whitepaper. 2001.

[25] Alon Peled. Index-Organized Tables: When should they be used? *SELECT Magazine*, pages 59–91, 2002.

[26] H.S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2002.

[27] WM Lam and JH Shapiro. A class of fast algorithms for the Peano-Hilbert space-fillingcurve. *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, 1, 1994.

[28] Jean Ihm, Xavier Lopez, and Siva Ravada. Oracle Spatial 11g: Advanced Spatial Data Management for Enterprise Applications. 2007.

[29] Oracle GeoRaster Java API. `http://www.oracle.com/technology/software/products/spatial/index.html`, As of March 19 2008.

[30] Jeffrey Xie Zhun Li Terry Xu. Oracle Database 10g GeoRaster: Scalability and Performance Analysis. 2005.

[31] Chuck Murray. Oracle Spatial: GeoRaster Developer's Guide. 2007.

[32] Hibernate Spatial. `http://www.hibernatespatial.org/`.

[33] Vivid Solutions - JTS Topology Suite. `http://www.vividsolutions.com/jts/jtshome.htm`.

[34] Open GIS Consortium. `http://www.opengeospatial.org/`.

[35] S. Šaltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez. Indexing the positions of continuously moving objects. *ACM SIGMOD Record*, 29(2):331–342, 2000.

[36] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *Information Theory, IEEE Transactions on*, 15(6):658–664, 1969.