Performance Impact of Streaming Doppler Radar Data on a Geospatial Visualization System

Beth Plale

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
beth@cc.gatech.edu

GIT-CC-01-07

Abstract

A geospatial visualization tool that is capable of visualizing weather data in conjunction with the terrain and object data typical of GIS visualization tools provides an important advancement in the study severe weather formation by providing a tool for weather scientists to study of the impact of land topography and human changes. In particular, unlike the plains region, tornados in the Southeast form quickly, are generally small, and are short lived. Weather scientists would like to understand the impact that high human population density areas, such as metro Atlanta, and topography have on tornado formation. Such an understanding could lead to better and earlier tornado predictions.

The visualization of doppler radar data raises interesting questions for the particular geographic visualization tool we are working with, the Virtual Geographic Information System (VGIS). In particular, doppler radar data is streaming, that is, it arrives as a continuous stream in near real-time. GIS visualization sytems such as VGIS are often designed under the assumption that the data to be visualized is relatively static. In this paper we assess the performance impact of visualizing doppler radar with VGIS, and propose a novel data structure and indexing scheme that reduces the performance overhead of retrieving and visualizing streaming data.

1 Introduction

VGIS, Virtual Geographic Information System [2], is a virtual environment for navigating large, high resolution geographic data sets. The system allows the user to navigate data sets containing terrain, buildings, roads, and high-resolution imagery draped on the terrain from any viewing angle. The project has of late been addressing the visualization of streaming doppler radar weather data received from live feeds from several local doppler radars. The project is also undertaking a port of VGIS to wireless clients.

In this paper we address the scalability of VGIS in the evolving environment of concurrent users, users with limited bandwidth and memory resources, and the receipt of streaming data. We argue that two key structural changes are needed in order for VGIS to scale to meet the demands of the evolving system As detailed in the paper, the first key change proposed is the integration of the two major run-time data storage structures: the run-time doppler radar tree and the spatial data tree. One contribution of this paper is the proposal of a novel integrated tree structure for temporal data and geospatial data that supports efficient update and retrieval for the types of requests we anticipate. The second proposal is that the long term,

physical storage structure must reflect the run-time tree structure. The existing dichotomy between logical structure and physical structure is the single largest roadblock to scalability that faces the VGIS system.

A second contribution of this paper is the unifying notion of a *weather event* as a human-identifiable and nameable weather activity (*e.g.,* 'Atlanta Tornado of 30 April 2001'). 'Weather events' are compositions of sequences of doppler scans from one or more doppler radars. A particular doppler scan belongs to a 'weather event' if the scan is performed while the weather event is active.

'Weather events' are elegant: they form a unifying basis around which we filter, organize, and query the weather data.

- Filtering: we filter (discard) data that doesn't participate in a weather event. This can greatly reduce the 15,000 file/day load on the vis client.
- Organization: timesteps are grouped according to the weather event to which they belong.
- Querying: once we start collecting months and months of data, the user is going to need a logical (human understandable) way of requesting data. Using the notion of 'weather event', the user can ask to view a particular event (in addition to asking to start at some timestep). Since the notion of weather event is reflected in the data organization, retrieving all timesteps for all radars associated with a weather event is efficient.

The remainder of the paper is organized as follows. In Section 2, we introduce the doppler radar data. Section 3 discusses some fundamental notions to multidimensional data storage and retrieval. In Sections 4, 5, and 6 we discuss the current logical storage representation for spatial and temporal data and propose a new structure with better efficiency. In Section 7, we turn attention to the physical storage representation and show the impact of the dichotomy that currently exists between physical and logical representations. Section 8 compares how runtime arrival of data events are handled in both the existing system and under the proposed integrated data storage that is unified across physical and logical representations. We conclude with a summary in Section 9. Appendix A contains a detailed description of the doppler radar data.

2 Run-time Doppler Radar Data.

The temporal data we are interested in is doppler radar data obtained from three radar sites in the Southeast: Peachtree City, GA, Grier, SC, and Hytop, Al. A doppler radar continuously scans, creating 3D volumes. A volume consists of 360° scans at 7-9 elevations ranging from 0.5° to 19.5°. It takes roughly 5-7 minutes to complete. The data product of a volume is collectively known as *radial data,* or more colloquially as Level 2 data, and consists of the following:

- radial velocity (veli): measure of the speed of an object as it moves either toward or away from the radar. Radar reads an object moving perpendicular to it as having velocity of 0. Velocity is positive for objects moving away, and negative for objects moving toward the radar.
- reflectivity (dbzi): amount of electromagnetic energy returned to the radar from a receiving object. The larger the object, the greater the returning energy. The scale is logarithmic.
- spectral width (spwi): measure of spread of radial velocity. That is, the spread of velocities of the objects within a cell. Agitated particles within a cell are moving in random directions. The spectral width captures this randomness. A high value (spread) is considered an indication of interesting weather behavior.

Radial data for a single scan is available immediately upon completion. The radial data collected over a 24 hr. period ranges between 200Mb and 400Mb, with larger sizes associated with increased weather intensity. For purposes of this paper, it is sufficient to think of the weather data as a stream of records continuously arriving at the data store. A *record* corresponds to a single sweep for one variable generated by one radar at one time instant. It the smallest unit of information generated by the WDSS system. A

record exists as a file, one record per file. For instance, the record `KFFC25May01_NXdbzi199.1.gz` is reflectivity data at timestep 199 of 25May01 generated by radar KFFC (Peachtree City, GA). A record is roughly 31K in size[1]. A *scan* is a collection of sweep records for a single radar. A radar makes 7-9 sweeps at different altitudes to complete a scan. A scan, which we equate to a logical time instant, completes in roughly 5-7 minutes of wallclock time. Thus 230 scans are completed per day. A scan consists of velocity, reflectivity, and spectral width records for each sweep in a scan, one record per species. A scan consists of roughly 837K of binary compressed data in 27 files. *Timestep* is the broadest organizing concept. A timestep is a logical unit of time consisting of a set of scans, with the maximum number of scans being one per radar. A timestep consists of velocity, reflectivity, and spectral width records for every radar participating in the timestep. Since in practice the radars do not behave in lockstep, there is no- correspondence between a timestep and wallclock time. For instance, timestep 214 was generated at 8:00 a.m. 07 May from the Peachtree City radar, at 6:50 a.m. 07 May from the South Carolina radar, and 7:44 a.m. from the Alabama radar. A timestep is roughly 2500K of binary compressed data. Assuming three radars, timestep can consist of up to 81 records. A *day* consists of 230 timesteps, or roughly 577MB of binary compressed data in 17,415 files. See the appendix for details on the doppler radar data.

3 Fundamentals of Multidimensional Data Storage and Retrieval

Efficient data retrieval from a tree-like structure has undergone extensive study by the database community. The community has assessed numerous tree-like structures for their efficiency in the against the types of requests most likely to be asked. In order to leverage this large body of understanding, we cast the VGIS system with its need to extract portions of the data store for purposes of visualization as a general data retrieval problem against a tree-like structure. In this way we can evaluate a particular storage model by its efficiency in retrieving and updating data.

Choice of logical storage representation is driven by the dimensionality of the indexes, the types of requests made, and the frequency of updates. Tree-like structures are used for multidimensional data,that is, data wherein the index, or key, is formed from more than one attribute. Put another way, multidimensional data exists in a 2-D space or higher. The geospatial data is indexed on two dimensions, latitude and longitude[2] This is to say that a record, or image, or elevation plot, can be uniquely identified solely by its latitude and longitude coordinates.

The temporal data is also multidimensional, but in a 4-D space. Latitude and longitude are sufficient to distinguish a record as having been generated from one radar versus another. But timestep is needed to distinguish between radar scans generated by one radar. We propose a fourth index that identifies a radar scan as belonging to a particular 'weather event'. The *weather event* is a human-identifiable and nameable weather activity (*e.g.*, Atlanta tornado 30 April 2001'). Weather events are compositions of sequences of doppler scans from one or more doppler radars. A particular doppler scan belongs to a 'weather event' if the scan is performed while the weather event is active. Weather events serve to aid user cognition by providing a named event. The major strength of the additional index is the ability it provides for efficient retrieval because records are indexed on and organized around the weather event.

Tree structures are well suited for multidimensional data. Which among the various types of tree structures is most efficient for our needs depends upon the types of requests anticipated. Requests for data stored in a database are referred to as queries. We identify several relevant query classes [3] and show that the requests made by the visualization engine can be cast into one of these general query classes. The three relevant classes are:

[1] Ranges from 9K to 70K.

[2] We ignore for a moment elevation which is an index attribute appearing only at the leaf level.

- *range query* - a request for data in a region of the grid. All points found in the buckets that cover that region will be answers to the query.
- *nearest neighbor query* - specifies a point P and asks for the closest point to P.
- *where-am-I query* - specifies a point P and asks for the data region or regions in which the point lies.

We maintain that these three query classes are either currently being supported by VGIS or will soon need to be supported under extensions to the VGIS system. Specifically, a range query occurs when a user initiates vertical flight towards the earth's surface, targeting the region at the center of his view. The descent can be viewed as a series of queries issued for increasingly higher resolution data of an increasingly smaller region. The number of queries issued is dependent upon the number of levels of detail supported. A nearest neighbor query occurs when a user initiates horizontal flight at an elevation above the earth's surface and in a vector from one region to another. The 'where-am-I' query is new, prompted by the relatively recent extension in VGIS to users of portable computing devices. Suppose a user has GPS installed in their portable device or on their person, so that the device can be cognizant of its specific location. Portable devices have limited storage resources, so are restricted to querying for data for the localized region around their current position. This is an example of a 'where-am-I' query. [3] These simple query descriptions belie the fact that there is a myriad of hard work in detail management, rendering, tracking, and navigation taking place simultaneously.

We posit that three types of queries must be supported on the temporal data. Though the query classes for temporal data overlap in two cases with the geospatial queries, the index attributes involved differ give the queries different semantics:

- *range query* - the range query is a request for data ranging over a set of timesteps. All points found in the buckets that cover that range of timesteps will be answers to the query.
- *nearest neighbor query* - movement through timesteps.
- *partial match query* - specify values for one or more dimensions and look for all points matching those values in those dimensions.

The range query exists for queries of the type "play the period 01 January to 31 January". There is an assumed region in this query, the region associated with the user's current position in the quadtree. The partial match query is a query for a specific weather event, for instance "play the Hurricane Andrew". Finally, a nearest neighbor query example would be "Start at timestep 01 January 10:07 a.m. 2001 and give me the next 30 timesteps (roughly 4 hours)".

In this section we have shown that the VGIS data storage problem can be cast into the database problem of querying multidimensional data. We point out the fundamental difference between geospatial data and the temporal data in the degree of multidimensionality and the types of queries specified. It is these fundamental differences that drive a solution to the data storage problem for temporal data that differs from that chosen for geospatial data.

4 Existing Logical Data Storage Structures

The logical structure adopted by VGIS for spatial and object (*i.e*, building) data storage is the quadtree. Each interior node in a quadtree corresponds to a square region in two dimensions, and has four children corresponding to its four quadrants. The quad tree is a suitable choice for storing geospatial data because the data is reasonably uniformly distributed. Quad trees have a fixed notion of child regions, that is, the region of a child is fixed at 1/4 the region of the parent. Uniform distribution of data assures a relatively even distribution of grid points into quadrants. A grossly uneven distribution results in wasted storage space. The

[3] Note that when faced with a where-am-i query under the current logical storage structure, the portable device will be required to provide the 'zone' in which it is currently situated. 'Zone' is purely an implementation artifact; the knowledge of it should not be forced upon every possible portable involved.

authors concur that the quad tree representation is well suited for the visualization of geospatial data because it provides efficient access to data for range queries, nearest-neighbor queries, and where-am-i queries.

Tree depth is determined in part by the number of altitudes at which image data must be available, as well as by the data resolution at the lowest levels. Interior nodes are annotated with the detail information. The actual representation used is a set of quadtrees indexed by a zone number. A zone number is a number between 1 and 32 that corresponds to the 32 segments into which the earth's surface is divided. Each segment node is then the root of a quad tree. At mid-level depths, octrees are employed to reduce the path length.

The logical structure for the temporal data is similar in design to that for the spatial data: the quadtree. More precisely, the tree is a mix of quad nodes, octree nodes, and what are referred to as volumes, but we refer to the entire structure as a quadtree because it preserves the essential feature of a quadtree: that of a fixed division of a region into equivalently sized child regions. The quadtree, that like its spatial cousin is built at startup, consists of approximately 17-18 levels, employs quad nodes at the top levels, octree nodes at the lower levels, and volumes at the leaves. The *volume* as used here is a 3-D index of latitude, longitude, and elevation, composed of 16x16x16 *bins*, each bin covering a region of 350m x 350m and with an elevation of 170m. When a timestep arrives at the quadtree, the tree is traversed to locate the appropriate volume, and the timestep is broken apart and distributed amongst the bins in the volume. The timestep fragment is added to the bin by inserting it at the front end of a linked list that stores all timestep fragments associated with the bin. The presence of a new timestep is then percolated up the quadtree by executing a percolation-style algorithm.

The limitations of the storage structure are severe: first, a linked list for holding timestep fragments is highly inefficient for the non-optimized operation. Since the structure has been optimized for update, that is, new timestep fragments are added to the front of the list, the performance impact is on requests. The request for a single timestep fragment requires $O(n)$ path operations where n is the number of timesteps stored. Retrieving a timestep fragmented over C = 16x16x16 = 4096 bins requires $O(n + C)$ operations where C is nontrivial. Second, the queries that we must support, particularly, the partial match query to retrieve a named weather event, cannot be performed efficiently on the existing store since weather event is not an indexed field. Even if it existed as an attribute to the current record, retrieval of a weather event would require $O(n)$ path operations. Third, the strategy of breaking a timestep into fragments that are distributed across the 3-D volume leaf node is problematic. Part of the problem lies in the existing format conversion algorithm that expands the data by a factor of 5 over its original size. That is, a record is anywhere between 9K and 70K when stored as binary, compressed data. Expanded it occupies anywhere between 45K and 350K. The difference over a day is large: the three radars together export roughly 577MB per 24hr period as compressed binary which is 2.9G in uncompressed ascii. Were a compressed representation found for the data, the quantity of data added to a bin per record would amount to no more than a few bytes per bin. At an anticipated arrival rate of one record every four seconds, the overhead of this preprocessing step is non-trivial.

One might argue that the extra processing overhead of fragmenting a record will be absorbed by creating another thread to handle weather data insertion. That argument holds as long as the underlying machine has idle processors. But VGIS already employs 6 threads, and the trend is toward porting the system to smaller machines, not larger. Laptops, PCs, and portable devices have one or two processors maximum, so there is no place to hide the overhead. In a system already taxing its computational resources, the additional overhead will result in reduced frames per second; the final metric by which visualization system performance is evaluated.

5 Proposed Storage Structure for Temporal Data

We propose a multi-key index for storing temporal data based on the multidimensionality of the index attributes, the types of queries supported, and on the correlation of index values. A multiple-key index is a tree in which nodes at each level are indexes for one attribute. The indexes are, in descending order from the root, weather event, timestep, and $min < latitude, longitude >, max < latitude, longitude >$ as shown in Figure 1. Weather event and the $< latitude, longitude >$ region are represented as hash tables, timesteps are a binary tree. The multiple-key index has the property that additions or retrievals of a single timestep can be done in $O(\log_2 n)$ where n is the number of timesteps. Correlation of index values is an important consideration because some logical representations, in particular the quadtree, are very poorly suited to correlated indices. The quadtree requires a node branch to four children of equal size representing one quadtrant of the parent region. When the indexes are latitude and longitude, the data points are grid points that are likely to be uniformly distributed. But when the dimensions are weather event and timestep, a high correlation exists, so the data points follow a pattern which results in many empty children buckets and hence wasted space.

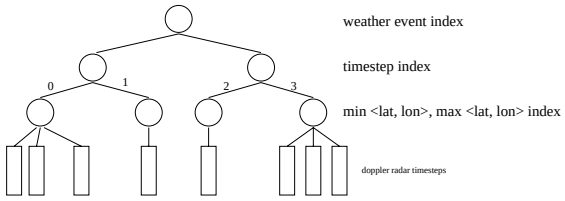

Figure 1: Multiple-key index for doppler radar data.

Partial match queries on a multiple-key index are very efficient. We use the root index to locate the weather event; data points are everything under. We make range queries efficient by using a B-tree for the timestep index. The range query (*e.g.,* "give me timesteps 01Apr01 to 30Apr01") requires searching all weather events in a region for timesteps in the range 01 Apr to 30 Apr. Inside a weather event, traversal is through the B-tree, which provides efficient access of timestamps in a range. Nearest neighbor queries can be cast as range queries with the following simple algorithm. To find the nearest neighbor of point t_0, we find a distance i such that we can expect to find several points within distance i of t_0. We then ask the range query $t_0 - i <= t <= t_0 + i$. If there turn out to be no points in this range, then we increase the range and search again.

We considered other logical storage structures for the time dependent data. Tree structures in general are good candidates for range queries and nearest-neighbor queries on multidimensional data. On the other hand, tree structures suffer from being deeper in some parts than others, making some traversals more inefficient. Our trees stay balanced because 1.) a weather event has a finite, usually small, number of timesteps, and 2.) the part that grows, the timesteps, is a binary tree for balance control.

The kd-tree (k-dimensional search tree) is a binary tree in which interior nodes have associated to them an attribute that splits the data points into two parts. Attributes at different levels of the tree are different, with levels rotating among the attributes of all index dimensions. The kd-tree is less suitable because every node is split on a boolean condition (true or false), which could lead to long paths for temporal data. Quadtrees are not recommended when two or more indexes display high correlation because the rigid division of a quadnode into fixed quadrants results in much wasted space when storing non-uniformly distributed data. As shown earlier, weather event and timestep are highly correlated. Hashing schemes can also be suitable for multidimensional indexed data. Partitioned hashing is a hash on multiple dimensions where each dimension contributes to the bucket number. Partitioned hashing has been shown to be ineffective for range queries

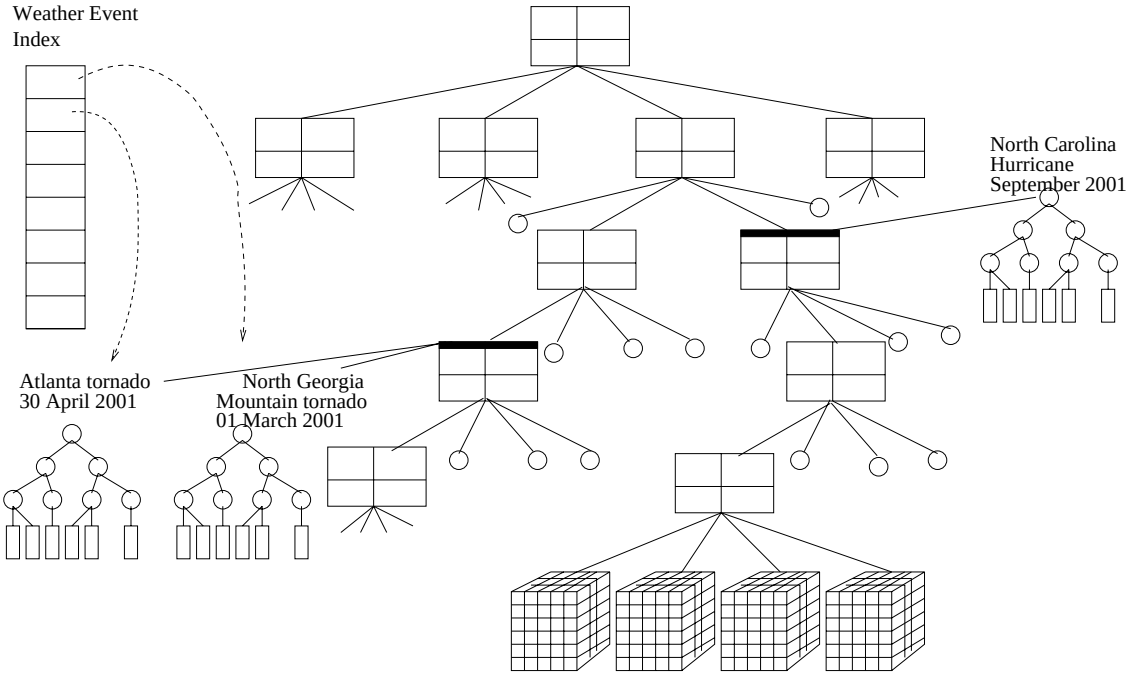

Figure 2: Integrated geospatial temporal tree

because the physical distance between bucket points is not reflected by the closeness of bucket numbers.

So of the index schemes considered, the multiple-key index is best suited for temporal data with index attributes showing high correlation, and for the types of queries that we need to support.

6 Integrated Logical Data Storage Structure

We propose an augmented geospatial quadtree for storing geospatial and temporal data for visualization. Specifically, we augment particular nodes in the geospatial quad tree with multi-key index trees for temporal data. The idea is based on the Time-Space Partitioning tree (TSP) [5]. TSP represents a time-varying volume both in temporal and spatial domain. The TSP tree is a time supplemented octree. That is, the skeleton of a TSP tree is a standard complete octree, which recursively subdivides the volume spatially until all subvolumes reach a predefined minimum size. Temporal information is stored in the TSP by associating with every octree node a binary tree that holds the time-based data. Every leaf node in the binary time tree represents the same subvolume in the spatial domain but a different timespan. Interior nodes of the binary time tree delimit a time span.

The TSP does not sufficiently address our needs for two reasons. First, TSP is based on the assumption that spatial and temporal data are static. That is, it is assumed that the TSP tree is constructed once then employed repeatedly. Our logical organization, on the other hand, must be flexible to accommodate arrival rates on the order of one event every 4 seconds. The second problem is that the TSP multidimensional index for the temporal data is latitude, longitude, and timestep. Since we cannot assume a user will run VGIS continuously to process the continuously arriving weather data, we need a scheme for storing and accessing threads of weather events. Thus our index is distinguished by its fourth dimension, the weather event.

We propose that the terrain quadtree be augmented with multiple-key index trees, one multiple-key tree per weather event; see Figure 2. There are numerous advantages to the organization. By combining the

spatial tree and the temporal trees, one eliminates the need to traverse two trees at runtime and maintain synchronization between the two. Second, the proposed structure allows $O(\log_2 n)$ access and retrieval to a timestep, compared to $O(n)$ for the existing design. Finally, it eliminates the costly step of partitioning timesteps and distributing fragments amongst bins at the leaf. To eliminate the need for traversing the terrain tree every time a timestep is received, we introduce a separate hash table on weather event to allow $O(1)$ path operation hops to a particular multiple-key tree. The hash table is shown in the upper left corner of Figure 2.

We additionally propose that timesteps be stored at the highest node in the tree in which the region encompassed by the weather event is completely defined by the quad node. This strategy reduces traversal cost, but more importantly, avoids the overhead of splitting weather events over multiple quadnodes. A tornado over Atlanta might be small enough to be completely captured by a single radar so resides at a lower quadnode than a hurricane which encompasses the entire Southeast coast, involving doppler data from radars located throughout the Southeast. See Figure 2.

7 Mapping Between Logical Data Store and Physical Storage

An important aspect of overall system performance is the mapping between a logical data storage structure and physical storage. That is, how the tree is stored to a set of files on disk. In a database, a tree would have a representation on disk that closely mirrors its logical representation, that is, the content of each quadtree node, for instance, would be preserved on disk, with the pointers serialized to some persistent notation. Reconstructing the quadtree would be a matter of replacing the serialized representation of child pointers with actual address pointers. Preserving the logical structure on disk has the advantage of allowing performance optimizations during tree building and traversal that are not possible in the current design.

7.1 Existing Physical Storage of Geospatial and Temporal Data

The physical storage organization of the VGIS data set bears no resemblance to a quadtree. The quadtree is a runtime artifact, constructed at startup from the dataset stored on disk, existing for the duration of the run, then destroyed. Specifically, the physical data storage organization is a hierarchy of object types. Top layer objects are building collection, terrain collection, and object collection. Within each collection are data types: imagery data, elevation data, and property data. Within each data type is a one-level organization of zones, that is, zones corresponding to a coarse division of the earth's surface into 32 regions. All data files, even the highest resolution 1m data, are partitioned amongst the 32 zones. An object type hierarchy of this type has several problems as a physical storage structure in support of a quadtree. First, additions to the quadtree at runtime require explicit knowledge of the underlying physical storage representation. Dynamic addition of a previously unknown building or the receipt of a new timestep not only must be integrated into the quadtree structure, but must be integrated into the underlying physical storage layout as well. Not having a separation of concerns results in more complex code. It also degrades visualization performance by requiring that the system reformat the data to a form needed for physical storage. A more viable option would be to update the quadtree at runtime when an event arrives then prior to program termination commit the quadtree to storage.

Second, the existing physical storage representation imposes significant startup overhead on the visualization tool. The quadtree must be constructed at startup. The interior nodes of the quadtree are populated with lower resolution data computed from the high resolution data of the leaf nodes. This percolation from leaves up to interior nodes requires that every leaf node be touched at startup. This point bears repeating. By building the tree from scratch at startup, one is required to touch every file in the data set. It may be argued that the quadtree structure is partially reflected in the stored 'properties' files, added during the off-line build

process that takes place when a new data set is added. Though the author may be in error, the build process is more likely to index the physical storage so that when VGIS builds the quadtree, it knows where to go to retrieve a particular data file. Suppose for a moment that the quadtree is partially built. What would then be reduced is the percolation time at startup. But the source code bears out the author's position that every file (that is, property, image, and geometry file) is touched at startup, so the problem still remains[4] The significant contributor to startup overhead is in the touching of every file.

Third, quadtree construction at runtime and the percolation strategy for populating interior nodes prohibits optimization strategies such as delayed loading of lower levels of the quadtree, careful packing of quadnodes into files to optimize retrieval under NFS, or selective loading of slices of the quadtree. As mentioned earlier, portable computing devices are memory constrained. The Compaq Ipaq, for instance, has 16M of memory. It is clear that taking a slice of the 3.2 G data set is one of the only viable options to supporting the devices. As we show in the next section, even high-end devices can no longer support the dichotomous storage strategy under the growth rate at which the data set is currently expanding.

Fourth, integrating time-dependent data into the existing bi-modal storage representations requires that the terrain manager, who is responsible for constructing the quadtree for terrain data, also build the multi-key index trees that hold the weather data. The additional logic required to construct and place the time-based nodes increases the complexity of the already complex terrain manager.

For these reasons it is important that the physical representation mirror the logical representation. Specifically, physical storage should consist of, in addition to the low level image files, files of persistent quadnodes. The quadnodes should hold the detail information about the lower levels in persistent store to preclude the need for percolation at startup. The quadnodes should be packed to files respecting 8K boundaries in order to take advantage of the 8K block size used by NFS for remote file transfer. Quadnodes should be grouped to a file based on spatial locality in the tree. As a simple example, nodes existing at different levels are not grouped together.

Temporal Data Storage Structure. The existing logical and physical organization of the time-dependent data experiences the same organizational dichotomy as the spatial data. The temporal quadtree is constructed at startup from a directory of timestep data organized in a flat hierarchy of timesteps. That is, all timesteps reside in a single directory.

7.2 Performance Assessment of Accessing Data Store.

VGIS relies on NFS, Network File System, to manage its remote file access. The authors agree that NFS has the advantage of good portability since academic institutions frequently use either NFS or Andrew, with NFS installations outnumbering Andrew significantly. An alternative would be to store the data set to a single large file that is managed by a parallel I/O management system. Data might be partitioned or file fragments interleaved across multiple disks [1]. But the reliance upon NFS does introduce a performance penalty for a number of reasons. NFS uses RPC on top of TCP for its communication protocol. Thus file transfers incur overhead attributed to the RPC protocol, XDR encoding and decoding, and clashes between the RPC protocol and TCP protocol that have become evident in WAN settings [4].

VGIS accesses files through the GSD file management system. GSD is a locally developed library for handling geographic data. A read call to the GSD library has the behavior of returning the contents of the entire file in a single read. Though this behavior is useful for managing image data that may not have natural boundaries, the implication is that one receives the entire contents of a file regardless of the amount of data

[4]There is consenses that every property node is touched. Disagreement exists as to the number of image and elevation files touched at startup. Regardless of the number of files in question, lengthy startup time is the primary scalability issue.

actually needed. The effect on performance is that of exacerbating the already problematic bottleneck that exists in transferring files from disk.

A typical VGIS configuration runs the visualization on a 64 bit, quad-processor, mips R10000 SGI located in CRB. The data set resides on a 64 bit quad-processor mips R12000 SGI located in CCB. The link between CCB and CRB is Gigabit Ethernet. The link between each machine and its network switch is 100 Mbps switched Ethernet. CNS currently guarantees an 8:1 over-subscription, and is moving toward a 4:1 over-subscription. An 8:1 over-subscription means that at most 8 100 Mbps ports feed a 1 GB line. So the theoretical maximum end-to-end bandwidth is 100 Mbps. But the realized bandwidth is likely to be significantly lower due to a number of factors. NFS uses 8K block sizes despite the GSD file interface's support of transferring the entire contents of a file on a single read. The XDR encoding and decoding and RPC incur overhead, and the data exists in 16,600 separate files. We estimate that a bandwidth of 25 Mbps is realistic. NFS caches recently read files so that subsequent accesses to cached files have a markedly faster access time. But the gains of NFS caching cannot be realized during startup because files are read only once.

Data set size (Gb)	*Bandwidth (Mbps)*	*estimated time to load data (secs)*
3	100	32
3	25	128
98	100	980
98	25	3920

Table 1: Data Access Time.

If we theoretically scale the data set size so that all 32 zones are populated at the same density as is found in the zone covering Atlanta, and we allow for a conservative one month of doppler radar data for three radars stored at any time, the data set size reaches 80G for spatial data and 18G for temporal data. Table 3 shows the startup times in seconds to transfer data from CCB to CRB for the current data set size at 3G and the anticipated size at 98G. The rates are the optimum transfer rate of 100Mbps and a more realistic 25Mbps. Read rates at startup on the order of 65 minutes are clearly unrealistic, but as long as the quadtree continues to be constructed from scratch, necessitating a touch of all leaf-level data files to populate the interior nodes, the times are unavoidable. One could argue that decreased times can be obtained by upgrading the 'last mile' links to gigabit Ethernet. Realistic throughput with gigabit Ethernet are running around a startling 250 Mbps, so the gain is not as dramatic as one might think. Too, recall that the shift is toward portable devices where the bandwidth at best is 100 Mbps, with 10 Mbps more likely.

8 Processing New Data

Currently two kinds of data can arrive at runtime: objects from exploratory clients who update the VGIS data store with information not currently recognized in the terrain data, and doppler radar data. As described earlier, the elegant approach is to store a new piece of data to the integrated spatial/temporal tree, then prior to termination commit the tree to permanent storage by serializing the objects to persistent store. In this way a new data object is stored to the logical data store on-the-fly and to physical store for a delayed write in one action. This proposal, by the way, does not preclude an off-line approach for adding large segments of geospatial data to the dataset.

Suppose instead that one is forced to live with the existing dichotomy between physical and logical storage representations. What would processing an incoming data stream look like? The 'build' process indexes the physical storage structure. That is, it encodes information about the physical storage structure

into the properties files so that when the VGIS system builds the quadtree at startup, it knows where the data files it needs are located. This approach is sufficient for static data but is problematic for data arriving in real time. To begin with, the build process is run off-line. But it could be modified to run as a daemon. Required too would be modifications to enhance the types of formats accepted: images, ascii text, GSD, socket messages, etc. When an event arrived, the VGIS system could store the event to its logical data store then pass the event on to the build process daemon which would, in turn, store the event to the physical data store. The event would be available upon the next VGIS system startup. But this solution does not address the mounting scalability problem at startup. For that problem to be solved, the two storage models must come together.

9 Conclusion

We argue in this paper that two key changes are needed in order to realize scalability growth that the VGIS visualization system demands. The storage structure for the real-time doppler weather data must be integrated into the spatial data store. This is for several reasons: an integrated structure eliminates the overhead of synchronizing between and traversal of two trees. It allows $O(\log_2 n)$ access to any timestep instead of $O(n)$. Finally, it removes the need to partition timesteps and distribute the fragments between bins. Second, the physical storage structure must reflect the logical storage structure. This enables adding records on-the-fly because the runtime system, which knows the quadtree structure, can simply commit a delayed write to have the new record reflected in storage. Right now, adding records requires an off-line process be executed on a staid data structure. It would greatly decrease startup times because percolation and global file touching would not be needed, Finally, it allows partial reading of quadtree (upper levels) which will greatly lessen the number of files touched.

10 Acknowledgments

The author would like to thank Zac Wartell and Mitch Perry for extensive discussions about the existing data structures.

References

[1] T. W. Crockett. File concepts for parallel I/O. In *Supercomputing 1989*, November 1989.

[2] Nickolas Faust, William Ribarsky, T.Y. Jiang, and Tony Wasilewski. Real-time global data model for the digital earth. In *International Conference on Discrete Global Grids*, 2000.

[3] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.

[4] Richard P. Martin and David E. Culler. NFS sensitivity to high performance networks. In *ACM SIGMETRICS*, Atlanta, GA, May 1999.

[5] Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu Ma. Time-varying volume rendering using a time-space partitioning (TSP)tree. In *IEEE Visualization*, October 1999.

A Appendix

Doppler radar data is obtained from three NEXRAD radars located at: (1) Peachtree City, GA, KFFC; (2) Grier, SC, KGSP; and (3) Hytop, AL, KHTX.

The following provides some idea of the types of files produced by the wdss system and typical compressed file sizes. The information that follows may not be complete (i.e., some wdss files not represented, sizes for some file types missing, acronyms not defined, etc.,). Of the files accumulated over a 24-hour period, some are generated after every volume and some after every scan. There are as many as nine scans per volume. Among those generated after every scan are the following:

Product	*Description*	*estimated size*
dbzi	reflectivity image	10k to 30k bytes
spwi	spectrum width image	10k to 30k bytes
imgh	image header file	up to about 1000 byte
veli	velocity image	10k to 130k bytes

Table 2: Radial Data Products.

Among those generated after every volume follow. Some of file types are generated under very specific weather conditions. That is, not all files listed below are generated for every volume.

Product	*Description*	*estimated size*
2dmeso	Mesocyclone Alg Overlay	600 bytes to 3k bytes.
alarm	Warning Alarm Message	32 bytes
bwer	Updraft Overlay	100 bytes.
cell		10k to 60k bytes
cmpi	composite reflectivity image	10k to 60k bytes
cmpo	Cell table information	100 bytes to 10k
cono	Cone of Influence Overlay	
hailt	HDA parameter trends	up to about 10k bytes
meso	Mesocycle detection algo (MDA) display output	
mesot	MDA parameter trends	
mesxs	MDA time-height cross-sections	
pcp1i	One-hour precipitation accum. image	2k bytes
pcp3i	Three-hour precipitation accum. image	2k bytes
pcpti	Storm-total precipitation accum. image	
scit	SCIT display output	
scitxs	SCIT time-height cross-sections	
tdat	TDA parameter trends	
tdl		100 to 900 bytes
tvs	TDA display overlays	50bytes to 750bytes
vadi	VAD (VWP) image	up to 10k bytes
vili	VIL (volume integrated liquid) image	1k to 3k bytes

Table 3: Analysis Data Products.

Description of doppler radar data Level 1 and Level 2 Sweeps are taken at elevations of:

- 0.5 °

- 1.5 °
- 2.4 °
- 3.3 °
- 4.3 °
- 6.0 °
- 9.9 °
- 14.6 °
- 19.5 °

The first sweep is slightly off the ground (not at 0.0 degrees) to cut down reflectivity.

The doppler radar generates Level 1 data. The radar generates a pulse every 1.5 microseconds at .5 watts. The antenna stays in position long enough to hear back from objects as far away as 150km. From the return stream, it samples 621 samples (from 0 meters to 150 km). The antenna sweeps 360 degrees then rises to its next elevation, sending out pulses on the way. Level 1 data is generally used for diagnostics only.

Level 2 data is processed Level 1 data; processed by a signal processor at the radar to generate reflectivity, spectral width, and velocity data. These collectively are referred to as 'radial data'. WDSS, Weather Decision Support System, located at NSSL, the Cobb County facility, and with Nick Faust, generates the three radial data products (perhaps slightly modified from the pure Level 2 data generated by the radar), and a number of metadata products (trends, overlays, messages) - a description of which I am currently seeking. Description of the main products is below:

- Radial velocity (veli): measure of the speed of an object as it moves either toward or away from the radar. Radar reads an object moving perpendicular to it as having velocity of 0. Velocity is positive for objects moving away, and negative for objects moving toward the radar.
- Reflectivity (dbzi): amount of electromagnetic energy returned to the radar from a receiving object. The larger the object, the greater the returning energy. The scale is logrithmic.
- Spectral width (spwi): measure of spread of radial velocity. That is, the spread of velocities of the objects within a cell. Agitated particles within a cell are moving in random directions. The spectral width captures this randomness. A high value (spread) is considered an indication of interesting weather behavior.

What follows is a detailed 24 hr. snapshot of the data from the Peachtree City, GA doppler radar taken on March 9, 2000. There are 32 products generated by the WDSS system. Acutal file listings are truncated to preserve readability. A summary of each product is given that includes its generation rate, numbering, number of files generated, and approximate range of file size.

```
Detailed snapshot of pcwdss1.gtri.gatech.edu/mnt/wdss/archive taken 8:14 a.m.
on March 9, 2000.
--------------------------------------------------------------------------------
1. Generated every 2 minutes from March 8 9:50 to March 9 8:14 (720
files, most at 27 bytes, 3 at 150 bytes)

-rw-r--r--   1 wdss     rts           27 Mar  8 09:50 0103081450.ctab.gz
-rw-r--r--   1 wdss     rts          150 Mar  9 02:32 0103090732.ctab.gz
-rw-r--r--   1 wdss     rts          150 Mar  9 02:34 0103090734.ctab.gz
-rw-r--r--   1 wdss     rts          150 Mar  9 02:36 0103090736.ctab.gz
-rw-r--r--   1 wdss     rts          150 Mar  9 02:38 0103090738.ctab.gz
-rw-r--r--   1 wdss     rts           27 Mar  9 08:14 0103091314.ctab.gz


2.  NX2dmeso - generated every 6 minutes.  Numbered 1.0 to 158.0.
Ranging from 368 bytes to 430.  158 files.
-rw-r--r--   1 wdss     rts          368 Mar  8 10:05 NX2dmeso1.0.gz
```

```
 -rw-r--r--   1 wdss     rts          368 Mar  9 08:12 NX2dmeso158.0.gz
```

3. NXalarm1 - generated every 6-10 minutes. Numbered 1.0 to 158.0. Either 20 bytes or 157. 158 files.

```
-rw-r--r--   1 wdss     rts           20 Mar  8 10:05 NXalarm1.0.gz
-rw-r--r--   1 wdss     rts          157 Mar  9 08:12 NXalarm158.0.gz
```

4. NXbwer - generated every 6-10 minutes. Numbered 1.0 to 158.0. From 27 to 34 bytes.

```
-rw-r--r--   1 wdss     rts           27 Mar  8 10:07 NXbwer1.0.gz
-rw-r--r--   1 wdss     rts           34 Mar  9 08:12 NXbwer158.0.gz
```

5. NXcandBwer - generated every 6-10 minutes. Numbered 1.0 to 158.0. 20 bytes.

```
-rw-r--r--   1 wdss     rts           20 Mar  8 10:07 NXcandBwer1.gz
```

6. NXcell - generated every 6-10 minutes. Numbered 1.0 to 158.0. Roughly 36700 bytes each.

```
-rw-r--r--   1 wdss     rts        36756 Mar  8 10:05 NXcell1.0.gz
-rw-r--r--   1 wdss     rts        36450 Mar  9 08:12 NXcell158.0.gz
```

7. NXcmpi - generated every 6-10 minutes. Numbered 1.0 to 158.0. bytes each. Anywhere from 14000 to 37000 bytes.

```
-rw-r--r--   1 wdss     rts        13612 Mar  8 10:05 NXcmpi1.0.gz
-rw-r--r--   1 wdss     rts        36560 Mar  9 08:12 NXcmpi158.0.gz
```

8. NXcmpo - generated every 6-10 minutes. Numbered 1.0 to 158.0. bytes each. 40 or 41 bytes.

```
-rw-r--r--   1 wdss     rts           40 Mar  8 10:05 NXcmpo1.0.gz
-rw-r--r--   1 wdss     rts           41 Mar  9 08:12 NXcmpo158.0.gz
```

9. NXcono - only 5 files, all generated early this morning. ??

```
-rw-r--r--   1 wdss     rts           59 Mar  9 02:21 NXcono100.0.gz
-rw-r--r--   1 wdss     rts           73 Mar  9 02:31 NXcono101.0.gz
-rw-r--r--   1 wdss     rts           59 Mar  9 02:40 NXcono102.0.gz
-rw-r--r--   1 wdss     rts           59 Mar  9 02:50 NXcono103.0.gz
-rw-r--r--   1 wdss     rts           59 Mar  9 02:56 NXcono104.0.gz
```

10. NXdbzi - generated every 1-3 minutes. Numberer 1.1 to 158.1 with 5-9 subentries. 4306 to 11000 bytes. Approx 800 files.

```
-rw-r--r--   1 wdss     rts        11302 Mar  8 09:56 NXdbzi1.1.gz
-rw-r--r--   1 wdss     rts         8006 Mar  8 09:59 NXdbzi1.2.gz
-rw-r--r--   1 wdss     rts         5511 Mar  8 10:00 NXdbzi1.3.gz
-rw-r--r--   1 wdss     rts         4306 Mar  8 10:02 NXdbzi1.4.gz
-rw-r--r--   1 wdss     rts         3695 Mar  8 10:05 NXdbzi1.5.gz
-rw-r--r--   1 wdss     rts        12130 Mar  8 10:07 NXdbzi2.1.gz
-rw-r--r--   1 wdss     rts         8398 Mar  8 10:10 NXdbzi2.2.gz
-rw-r--r--   1 wdss     rts         5877 Mar  8 10:11 NXdbzi2.3.gz
-rw-r--r--   1 wdss     rts         4654 Mar  8 10:13 NXdbzi2.4.gz
-rw-r--r--   1 wdss     rts         3940 Mar  8 10:14 NXdbzi2.5.gz

-rw-r--r--   1 wdss     rts        26004 Mar  9 08:07 NXdbzi158.1.gz
-rw-r--r--   1 wdss     rts        28700 Mar  9 08:08 NXdbzi158.2.gz
-rw-r--r--   1 wdss     rts        23261 Mar  9 08:08 NXdbzi158.3.gz
-rw-r--r--   1 wdss     rts        20420 Mar  9 08:09 NXdbzi158.4.gz
-rw-r--r--   1 wdss     rts        18262 Mar  9 08:10 NXdbzi158.5.gz
```

```
-rw-r--r--   1 wdss     rts        15170 Mar  9 08:10 NXdbzi158.6.gz
-rw-r--r--   1 wdss     rts        10910 Mar  9 08:11 NXdbzi158.7.gz
-rw-r--r--   1 wdss     rts         8261 Mar  9 08:11 NXdbzi158.8.gz
-rw-r--r--   1 wdss     rts         6808 Mar  9 08:12 NXdbzi158.9.gz
-rw-r--r--   1 wdss     rts        25735 Mar  9 08:13 NXdbzi159.1.gz
-rw-r--r--   1 wdss     rts        28445 Mar  9 08:14 NXdbzi159.2.gz
-rw-r--r--   1 wdss     rts        22932 Mar  9 08:14 NXdbzi159.3.gz

29 April data from KFFC (Peachtree City)

-rw-r--r--   1 beth     vrgis      32566 Apr 29 10:56 NXdbzi1.1.gz
-rw-r--r--   1 beth     vrgis      21803 Apr 29 10:59 NXdbzi1.2.gz
-rw-r--r--   1 beth     vrgis      15179 Apr 29 11:00 NXdbzi1.3.gz
-rw-r--r--   1 beth     vrgis      12115 Apr 29 11:02 NXdbzi1.4.gz
-rw-r--r--   1 beth     vrgis      10256 Apr 29 11:03 NXdbzi1.5.gz

unzipped sizes; files are of type 'data' (binary data)
cancer% file NXdbzi1.1
NXdbzi1.1:      data

-rw-r--r--   1 beth     vrgis      73446 Apr 29 10:56 NXdbzi1.1
-rw-r--r--   1 beth     vrgis      50822 Apr 29 10:59 NXdbzi1.2
-rw-r--r--   1 beth     vrgis      33864 Apr 29 11:00 NXdbzi1.3
-rw-r--r--   1 beth     vrgis      26356 Apr 29 11:02 NXdbzi1.4
-rw-r--r--   1 beth     vrgis      22004 Apr 29 11:03 NXdbzi1.5


11.  NXfsi - appears to be 1 file.
-rw-r--r--   1 wdss     rts          104 Mar  9 08:29 NXfsi0.0.gz

12.  NXhailt - generated every 6 minutes.  Numbered 1.0 to 158.0.
Roughly 3000 bytes.   158 files.
-rw-r--r--   1 wdss     rts         2909 Mar  8 10:05 NXhailt1.0.gz
-rw-r--r--   1 wdss     rts         2944 Mar  9 08:12 NXhailt158.0.gz

13. NXidx - appears to be one file.
-rw-r--r--   1 wdss     rts         1384 Mar  9 08:29 NXidx0.0.gz

14.  NXimgh - generated every 1-3 minutes.  Same characteristics as
NXdbzi.  Roughly 400 bytes.   Roughly 1000 files.
-rw-r--r--   1 wdss     rts          348 Mar  8 09:56 NXimgh1.1.gz
-rw-r--r--   1 wdss     rts          359 Mar  8 09:59 NXimgh1.2.gz
-rw-r--r--   1 wdss     rts          351 Mar  8 10:00 NXimgh1.3.gz
-rw-r--r--   1 wdss     rts          348 Mar  8 10:02 NXimgh1.4.gz
-rw-r--r--   1 wdss     rts          349 Mar  8 10:05 NXimgh1.5.gz


15.  NXmes88d - generated every 6 minutes.  Numbered 1.0 to 158.0.
Roughly 40 bytes.   158 files.
-rw-r--r--   1 wdss     rts           40 Mar  8 10:05 NXmes88d1.0.gz
-rw-r--r--   1 wdss     rts           41 Mar  9 08:12 NXmes88d158.0.gz

16. NXmeso -  generated every 6 minutes.  Numbered 1.0 to 158.0.
Roughly 40 bytes.   158 files.
-rw-r--r--   1 wdss     rts           40 Mar  8 10:05 NXmeso1.0.gz
-rw-r--r--   1 wdss     rts           41 Mar  9 08:12 NXmeso158.0.gz

17.  NXmesot -  generated every 6 minutes.  Numbered 1.0 to 158.0.
```

Roughly 900 bytes. 158 files.
```
-rw-r--r--   1 wdss     rts          885 Mar  8 10:05 NXmesot1.0.gz
-rw-r--r--   1 wdss     rts          891 Mar  9 08:12 NXmesot158.0.gz
```

18. NXmesxs - generated every 6 minutes. Numbered 1.0 to 158.0.
20 bytes. 158 files.
```
-rw-r--r--   1 wdss     rts           20 Mar  8 10:05 NXmesxs1.0.gz
-rw-r--r--   1 wdss     rts           20 Mar  9 08:12 NXmesxs158.0.gz
```

19. NXpcp1i - generated every 6 minutes. Numbered 1.0 to 158.0.
Roughly 1000 bytes. 158 files.
```
-rw-r--r--   1 wdss     rts          941 Mar  8 10:02 NXpcp1i1.0.gz
-rw-r--r--   1 wdss     rts          949 Mar  9 08:09 NXpcp1i158.0.gz
```

20. NXpcp3i - generated every 6 minutes. Numbered 1.0 to 158.0.
943 - 1800 bytes. 158 files.
```
-rw-r--r--   1 wdss     rts          943 Mar  8 10:02 NXpcp3i1.0.gz
-rw-r--r--   1 wdss     rts         1435 Mar  9 08:09 NXpcp3i158.0.gz
```

21. NXpcpti - generated every 6 minutes. Numbered 1.0 to 158.0.
939 - 1900 bytes. 158 files.
```
-rw-r--r--   1 wdss     rts          939 Mar  8 10:02 NXpcpti1.0.gz
-rw-r--r--   1 wdss     rts          942 Mar  9 08:09 NXpcpti158.0.gz
```

22. NXscit - generated every 6 minutes. Numbered 1.0 to 158.0.
Roughly 4000 bytes. 158 files.
```
-rw-r--r--   1 wdss     rts         3822 Mar  8 10:05 NXscit1.0.gz
-rw-r--r--   1 wdss     rts         3848 Mar  9 08:12 NXscit158.0.gz
```

23. NXscitxs - generated every 6 minutes. Numbered 1.0 to 158.0.
Either 20 bytes or 100-139 bytes; former more common. 158 files.
```
-rw-r--r--   1 wdss     rts           20 Mar  8 10:05 NXscitxs1.0.gz
-rw-r--r--   1 wdss     rts           20 Mar  9 08:12 NXscitxs158.0.gz
```

24. NXspwi - generated every 1-3 minutes. Same characteristics as
NXdbzi and NXimgh (14). From 8000 to 21000 bytes. Roughly 1000 files.
```
-rw-r--r--   1 wdss     rts        21627 Mar  8 09:56 NXspwi1.1.gz
-rw-r--r--   1 wdss     rts        15958 Mar  8 09:59 NXspwi1.2.gz
-rw-r--r--   1 wdss     rts        10605 Mar  8 10:00 NXspwi1.3.gz
-rw-r--r--   1 wdss     rts         7870 Mar  8 10:02 NXspwi1.4.gz
-rw-r--r--   1 wdss     rts         6413 Mar  8 10:05 NXspwi1.5.gz

-rw-r--r--   1 wdss     rts        43957 Mar  9 08:07 NXspwi158.1.gz
-rw-r--r--   1 wdss     rts        52038 Mar  9 08:08 NXspwi158.2.gz
-rw-r--r--   1 wdss     rts        46091 Mar  9 08:08 NXspwi158.3.gz
-rw-r--r--   1 wdss     rts        38466 Mar  9 08:09 NXspwi158.4.gz
-rw-r--r--   1 wdss     rts        33544 Mar  9 08:10 NXspwi158.5.gz
-rw-r--r--   1 wdss     rts        27412 Mar  9 08:10 NXspwi158.6.gz
-rw-r--r--   1 wdss     rts        19252 Mar  9 08:11 NXspwi158.7.gz
-rw-r--r--   1 wdss     rts        14333 Mar  9 08:11 NXspwi158.8.gz
-rw-r--r--   1 wdss     rts        11599 Mar  9 08:12 NXspwi158.9.gz
-rw-r--r--   1 wdss     rts        44587 Mar  9 08:13 NXspwi159.1.gz
-rw-r--r--   1 wdss     rts        53439 Mar  9 08:14 NXspwi159.2.gz
-rw-r--r--   1 wdss     rts        45379 Mar  9 08:14 NXspwi159.3.gz
```

25. NXsrm - generated every 6 minutes. Numbered 1.0 to 158.0.

35 bytes. 158 files.
-rw-r--r-- 1 wdss rts 35 Mar 8 10:05 NXsrm1.0.gz
-rw-r--r-- 1 wdss rts 35 Mar 9 08:12 NXsrm158.0.gz

26. NXtdat - generated every 6 minutes. Numbered 1.0 to 158.0.
Roughly 800 bytes. 158 files.
-rw-r--r-- 1 wdss rts 756 Mar 8 10:05 NXtdat1.0.gz
-rw-r--r-- 1 wdss rts 763 Mar 9 08:12 NXtdat158.0.gz

27. NXtdl - generated every 6 minutes. Numbered 1.0 to 158.0.
Roughly 100 bytes. 158 files.
-rw-r--r-- 1 wdss rts 106 Mar 8 10:05 NXtdl1.0.gz
-rw-r--r-- 1 wdss rts 109 Mar 9 08:12 NXtdl158.0.gz

28. NXvs - generated every 6 minutes. Numbered 1.0 to 158.0.
40 bytes. 158 files.
-rw-r--r-- 1 wdss rts 40 Mar 8 10:05 NXtvs1.0.gz
-rw-r--r-- 1 wdss rts 41 Mar 9 08:12 NXtvs158.0.gz

29. NXvadi - generated every 6 minutes. Numbered 1.0 to 158.0.
From 1600 to 8400 bytes. 158 files.
-rw-r--r-- 1 wdss rts 1499 Mar 8 10:05 NXvadi1.0.gz
-rw-r--r-- 1 wdss rts 4666 Mar 9 08:12 NXvadi158.0.gz

30. NXveli - generated every 1-3 minutes. Same characteristics as
NXdbzi, NXspwi(24), and NXimgh (14). From 10000 to 100000 bytes. Roughly 1000
files.
-rw-r--r-- 1 wdss rts 23904 Mar 8 09:56 NXveli1.1.gz
-rw-r--r-- 1 wdss rts 15786 Mar 8 09:59 NXveli1.2.gz
-rw-r--r-- 1 wdss rts 10777 Mar 8 10:00 NXveli1.3.gz

-rw-r--r-- 1 wdss rts 52892 Mar 9 08:08 NXveli158.3.gz
-rw-r--r-- 1 wdss rts 44399 Mar 9 08:09 NXveli158.4.gz
-rw-r--r-- 1 wdss rts 38722 Mar 9 08:10 NXveli158.5.gz
-rw-r--r-- 1 wdss rts 31464 Mar 9 08:10 NXveli158.6.gz
-rw-r--r-- 1 wdss rts 23319 Mar 9 08:11 NXveli158.7.gz
-rw-r--r-- 1 wdss rts 17991 Mar 9 08:11 NXveli158.8.gz
-rw-r--r-- 1 wdss rts 14688 Mar 9 08:12 NXveli158.9.gz
-rw-r--r-- 1 wdss rts 50500 Mar 9 08:13 NXveli159.1.gz
-rw-r--r-- 1 wdss rts 60305 Mar 9 08:14 NXveli159.2.gz
-rw-r--r-- 1 wdss rts 51806 Mar 9 08:14 NXveli159.3.gz

31. NXvili - generated every 6 minutes. Numbered 1.0 to 158.0.
From 60 to 111 bytes. 158 files.
-rw-r--r-- 1 wdss rts 111 Mar 9 05:05 NXvili126.0.gz
-rw-r--r-- 1 wdss rts 76 Mar 9 08:12 NXvili158.0.gz
(didn’t catch first record)

32. NXwarn - generated every 6 minutes. Numbered 1.0 to 158.0.
Either 25 bytes or around 85 bytes; most are the former. 158 files.
-rw-r--r-- 1 wdss rts 85 Mar 9 02:50 NXwarn103.0.gz
-rw-r--r-- 1 wdss rts 25 Mar 9 08:12 NXwarn158.0.gz
(didn’t catch first record)