GeoBase: Indexing NetCDF Files for Large-scale Data Analysis

Tanu Malik
Computation Institute
University of Chicago and Argonne National Laboratory.
Chicago, USA 60637
tanum@ci.uchicago.edu

ABSTRACT
Data-rich scientific disciplines increasingly need end-to-end systems that ingest large volumes of data, make it quickly available, and enable processing and exploratory data analysis in a scalable manner. Key-value stores have attracted attention, since they offer highly available data storage, but must be engineered further for end-to-end support. In particular, key-value stores have minimal support for scientific data that resides in self-describing, array-based binary file formats and do not natively support scientific queries on multi-dimensional data. In this chapter, we describe GeoBase, which enables querying over scientific data by improving end-to-end support through two integrated, native components: a linearization based index to enable rich scientific querying on multi-dimensional data and a plugin that interfaces key-value stores with array-based binary file formats. Experiments show that this end-to-end key-value store retains the features of availability and scalability of key-value stores and substantially improves the performance of scientific queries.

INTRODUCTION
Advances in remote sensing technology have significantly lowered the cost of data acquisition via satellites and aircrafts. Well-known satellites, such as GOES (NOAA), Landsat (NASA), and Aqua/Terra (NASA) continuously stream 20-60 GB of remotely sensed image datasets to receiving stations per day. To use these datasets for analysis, scientists typically determine relevant datasets through a metadata search. Given the large data volumes, a metadata search, however, is increasingly becoming insufficient in retrieving datasets of interest. Content-based searches, such as determining ranges of pressure or temperature within the data files, can improve retrieval. To conduct these searches, however, currently scientists have to download datasets, and perform data management tasks, such as format conversions, pre-process the datasets, index, and visualize, and then subset the content. It is not uncommon for scientists to work with many different systems to perform the entire data management task, Alternatively, it is desirable to have an *end-to-end* system that ingests datasets in their native format, indexes them as part of the ingestion process, and provides a simple API for content-based searches.

We are interested in designing such an end-to-end system for geoscience datasets. In several sub-domains of geosciences, end-to-end issues are addressed through data access libraries. For instance most geoscience datasets are stored in self-describing formats, such as NetCDF (Rew 1990), and HDF5 (Folk 1999), and are made available for exploration and analysis through the NetCDF-Java API or HDF5 API. These libraries address some end-to-end issues in that they hide the nitty-gritty details specific to their format. They, however, do not address other end-to-end issues. In particular, the libraries do not provide a mechanism for indexing the datasets or the ability to conduct content-based analysis on multiple datasets in parallel.

Another alternative is to explore data through parallel file systems tailored for NetCDF files (Li 2003), but these file systems introduce other end-to-end issues, namely, communication between high-level data analysis operators that must be controlled by the programmer (Buck 2011). Datasets can also be explored by using geospatial databases, such as PostGIS Raster and MySQL. Most database systems, however, require datasets resident in a file system to be imported into their native format. For instance, raster images can be ingested into the databases through the Geospatial Data Abstraction Library (GDAL), but bulk loading of large volumes of data is known to be slow. For analysis, the storage model in these traditional databases is row-oriented, i.e., they store multidimensional array data as a single record. They do not leverage the performance benefit derived from a column-oriented design, which has shown to provide orders of magnitude of performance improvement (Abadi 2008).

For geoscience applications, a useful alternative is distributed key-value stores, e.g., BigTable (Chang 2008) and their open-source counterparts, e.g., HBase (Apache 2010) that are built on a parallel storage framework and have shown to scale to millions of updates while being fault-tolerant and highly available. Key-value stores, however, do not address all end-to-end issues in data exploration of geoscience datasets. In particular, key-value stores do not natively support files in self-describing formats, which are commonly used in geoscience disciplines. They also do not provide for efficient exploration of multi-dimensional datasets. Lack of support for self-describing files leads to format transformation, and lack of support for multi-dimensional data leads to inefficient indexing, and additional data management that soon become too cumber- some given the size of common scientific data sets.

In this chapter, we present GeoBase an end-to-end system that enables efficient exploratory data analysis on geoscience datasets without incurring any additional data transformation costs. To achieve this vital goal, GeoBase uses space-filling curves to store multi-dimensional spatio-temporal data in a one-dimensional key-value store. In adopting space-filling curves as an indexing technique, GeoBase does not introduce any data transformation costs that are often implicit in the linearization process. GeoBase intercepts requests for the sub-queries on the space-filling curve and corresponds them to contiguous, low-level byte extents at the physical level. Through this modification, data can continue to reside in thousands of files without the need to transform and/or transport them to another system. Our work makes the following technical contributions:

1. The multi-dimensional index enables efficient data exploration through array operations, such as multi dimensional range queries and aggregation queries.

2. The underlying key-value store provides the ability to sustain large number of data files coming at a fast rate, ensuring fault-tolerance and high availability, but more importantly multi-dimensional exploration on spatial, time, and other dimensions.

3. The system can retain the original file formats and need not transform into alternate formats and/or make coordinate transformations. These formats are not legacy formats but chosen and widely adopted by the geoscience community as space efficient formats for describing data.

4. We conduct and present a thorough experimental study of our solution.

The rest of the chapter is organized as follows: In *Background*, we cover related work. We then describe the problems in designing an end-to-end system for geoscience applications in *Problems in End-to-End Analysis*. We then introduce an end-to-end key-value store for geoscience, the architecture of GeoBase and focuses on the storage layer and the indexing method. Our solution discusses optimization issues that arise in an end-to-end system and provides solutions for the same. In *Experiments* we describe a full evaluation of GeoBase through experiments done on thousands of NetCDF files taken from the climate modeling studies and using HBase as our key-value store.

BACKGROUND

In this section, we review some recent works that address aspects of end-to-end data management. In general, relational data processing improves end-to-end data processing by bringing analysis closer to storage. This is often achieved by abstracting the storage layer for analysis. Several early works abstracted data in the storage layer as multi-dimensional arrays and examined query-processing techniques (Libkin 1996, Marathe 1997). Baumann explored these abstractions on raster image data (Baumann 1999). More recently Howe and Maier proposed a logical grid abstraction by separating the topology of the grid from its geometry, which is encoded in the data files (Howe 2007). SciDB also improves end-to-end processing by natively enabling a variety of relational operators on multi-dimensional data (Stonebraker 2009). While these works improve end-to-end processing, none of them provide the scalability, ease-of-use and availability of key-value systems.

End-to-end issues have also been explored in key-value systems. In Zhao (Zhao 2010), the authors enable basic NetCDF processing via Hadoop but first require the data be converted into text. There has been work towards extending the NetCDF operator library to support parallel processing of NetCDF files to reduce data movement and transformation costs (Wang 2008), but does not address issues of efficient access. SciHadoop (Buck 2011) addresses fault tolerance issues and native access to scientific file formats but does not address multi-attribute analysis. Reducing the input space, i.e., using indexes in Hadoop systems has been investigated in HadoopDB (Abouzeid 2009) and in (Lu 2010). In both works, data must be ingested into the underlying database, thereby incurring extra data movement and preventing the use of scientific file formats, which are more commonly used.

Since scientific data is mostly read-only, another useful method for preserving locality is to construct bitmap indexes. FastBit, a multi-dimensional bit index, is built over NetCDF files (Wu 2009). Bit indexes, however, provide secondary, non-clustered index over the base data. Since geospatial data is multidimensional, an end-to-end system would benefit from spatial clustering, especially for range queries. Therefore, in this chapter we adopt space-filling curves (SFC) (Jagadish 1997) to cluster the data. We do not consider bitmap indexes but they can, in principle, be built over the clustered data. The size of the compressed bitmaps, in this case, will be smaller since clustering increases the average run length by storing tuples with equal values near to each other.

SFCs have recently been used to improve performance (Jensen 2004, Tao 2009), but must scale to large volumes of data. In this regard, a work closely related to ours is the one by Nishimura et al. (Nishimura 2011) on scalable multidimensional data infrastructure for location-aware services, such as location-aware social networking services or gaming services. However, our work differs from theirs in two primary respects. First, they primarily focus on object-based spatio-temporal data (point objects), whereas our approach exclusively focuses on field-based spatio-temporal data, in particular satellite imagery and its specific operators. Second, their linearization scheme requires data to be ingested in HBase, while we use Z-regions (Fenk 2002), a variant of Z-curve to provide native access to NetCDF files.

PROBLEMS IN END-TO-END ANALYSIS

We define an end-to-end system as a system that incurs minimal data transformation costs from the time it ingests data in a native format to the time data is available for exploratory analysis. Data transformation costs primarily arise due to differing choice of formats within the storage, intermediate, and analysis layers. Further, transformation costs may either correspond to one-time extract, transform, load (ETL) costs, which itself may be significant if the data comes in real-time and/or may correspond to repeated costs, especially if the same data is downloaded several times in a given format to be analyzed in a

different format later. In this section, we describe a typical data ingestion system for geosciences, popular types of analysis operations, and problems in conducting end-to-end analysis.

In geosciences, the data ingestion system is typically a file system storing datasets generated during an experiment, an observation or while simulating geophysical phenomena. Each dataset is a collection of data files, in which each file records multiple physical variables in self-describing, portable formats. Each variable is described as a multi-dimensional array and its data laid out in the file in a specified order. Figure 1 illustrates a data ingestion system.

Figure 1. Measurement of sea surface temperature and pressure variables on Earth. The continuous phenomenon is discretized into a regular grid; each grid cell records the values of the physical variables under its spatial extent, and is manifested into a file, based on a chosen format, such as NetCDF, HDF5, GRIB.

Two common exploratory analysis operations are:

Data Reduction: A data reduction operation subsets multiple geophysical regions in space and time. An example two-region query is "Project temperature and precipitation attributes for the selected region R1 and project solar radiation for the selected region R2 between a common time frame of 26th of June, 1990 and 26th of June, 2000." Typically there will be thousands of selected regions. For an analysis system a data reduction query corresponds to several multi-dimensional range queries.

Data Condensation: Condensation operations are data reduction operations followed by aggregation operations. The complexity of the aggregation operation varies from computing a simple maximum, minimum, or median to filter kernels, in which a new value for the array cell is determined by combining each old value plus its neighborhood. For the analysis system, condensation operation can be represented as a function f applied to a data reduction set S, yielding a result R, which is either scalar or composed of output of arrays $r \in R$. Thus,

$$f : (s_1, s_2, \ldots) \leftarrow (r_1, r_2, \ldots) \quad (1)$$

A data access library, such as the NetCDF API (Unidata) combined with data protocols, such as OPeNDAP (OPeNDAP) and Pomengranate (NASA) enable reduction operations over a data ingestion system and thus enable an end-to-end system. However, this end-to-end system does not scale for analysis over large number of files. We realized a modest end-to-end system with NetCDF API and OPeNDAP with datasets hosted in a Linux ext3 file system. Figure 2 shows the average total time it takes to execute 100 data reduction queries when the size of the dataset is increased from 500 to 27,000 NetCDF files. Each query asks for a subset of data by specifying a region over all files. The time to access increases non-linearly as in addition to serial execution on files, a large amount of state information is kept in memory. This strictly limits efficiency of data access, especially when the file system consists of millions of files or if the resolution of the file is increased.

Figure 2. End-to-end system performance of data ingestion system

Our objective is to build an end-to-end system on key-value stores that provides scalable, distributed access to data and thus improve performance. A vanilla key-value store currently does not lead to an end-to-end system. Most key-value systems support binary sequence file formats, which are different from self-describing formats of geoscience datasets. Thus data has to either translate into text or a custom translator needs to be written. Further, data reduction analysis operation is limited to one of the attributes (for e.g. one of the spatial dimensions) onto the key space of the key-value stores (Hill 2011). So analysis

operations such as data reduction and condensation cannot be performed efficiently on the same system where data is stored.

In this chapter, we determine if a more efficient end-to-end system can be designed that does not incur data transformation costs during the ingestion phase and also enables multi-attribute access so that reduction and condensation operations can be conducted on the data system itself and not by downloading datasets. Towards this, in this chapter, we explore enabling a native interface to self-describing datasets and a linearization technique as an indexing mechanism for multi-attribute access to build a more efficient end-to-end system. While a native interface reduces data transformation costs, linearization does not.

Linearization (Samet 2005) is a common technique to transform multi- dimensional data points to a single dimension and allows for efficient multi-attribute access in database systems. A space-filling curve (Jagadish 1997) is one of the most popular approaches for linearization, in which the curve visits all points in the multi-dimensional space in a systematic order. Hilbert curves and Z-ordering are examples of space-filling curves. Hilbert curve is a continuous fractal space-filling curve that preserves locality. Z-ordering loosely preserves the locality of data-points in the multi-dimensional space and is also easy to implement. However, linearization imposes data translation costs similar to those imposed by importing datasets in native format. In particular, to create linearized data all data values must be read from each individual file and reordered according the order of the curve.

Linearization also reduces data dimensionality to one and allows data to be searched in *log(n)* complexity. However, there is loss of information, often leading to false-positives. It is often possible that objects far away in the original dimension are close together in the reduced dimension. The converse is also true, *i.e.* objects near to each other in the original dimension may be further away in the reduced dimension. While several alternatives are suggested in literature (Jagadish 1997, Samet 2005), it is important that the chosen method is compliant with the native file system interface.

An End-to-End System

We address the shortcomings of key-value stores, described in the previous section, by introducing two capabilities in GeoBase: (A) a native interface for accessing datasets in self-describing formats, and (B) efficient multi-attribute access over the datasets. The addition of these features improves upon the much needed end-to-end functionality in key-value systems in that they provide efficient analysis over self-describing datasets, while continuing to provide high scalability and availability as new datasets are inserted. In this section, we first provide background on some technologies that are used in GeoBase. We then describe improvisations to the technologies and optimizations schemes in GeoBase.

Background

GeoBase relies on *(i)* NetCDF, the self-describing data formats widely used in geosciences, *(ii)* HBase, the open-source key-value store, and *(iii) Z*-curves, a simple and efficient linearization technique.

Self-describing formats

There are a variety of available formats in the geosciences, such as NetCDF, HDF5, GRIBB, Fast, etc. Amongst these formats NetCDF remains a widely adopted and popular format because of its simple data format and easy-to-use API. Consequently, in GeoBase we use NetCDF as the governing format.

Figure 3. NetCDF File Format. (Source: Li, Liao, Choudhary, et. al (Li 2003))

The NetCDF file format divides a given dataset file, physically, into two parts: file header and array data. The file header is the metadata defining *(i)* dimensions, *(ii)* global attributes, and *(iii)* variables. Dimensions such as spatial extents and time define shape of the variables in the dataset. One dimension can be unlimited and is considered the record dimension for growing-size variables. The header for the variables define its name, shape, named attributes, data type, array size, and data offset. The data part of the file, contains arrays of variable values (or raw data), and lays fixed-size arrays before laying out variable-sized arrays. Each array is laid out in row-major order. This ordering, as we show later, influences our choice of linearization.

An important characteristic of the geo-scientific data is that it is often skewed spatially. Skew may lead to certain regions being absent, which are handled as null values, and certain spatial extents being present at high resolutions.

An important characteristic of the geo-scientific data is that it is often skewed spatially. Skew may lead to certain regions being absent, which are handled as null values, and certain spatial extents being present at high resolutions.

Linearization

Z-ordering loosely preserves the locality of data points in a multi-dimensional space.

Functionally, *Z(p)* is a bijective function that computes for every tuple *p* its equivalent *Z*- address, which is a position on a linear *Z* shaped curve.

Definition (Z-curve by calculation)

$Z(p)$ = bitinterleave$(p_1,...,p_d)$,

in which $p_1 , . . . , p_d$ are d dimension values of tuple p. Figure 4 presents the *Z*-addresses for each point in a 4×4 grid. *Z*-addresses are efficiently computed by bit-interleaving, i.e., by interleaving the bit-representation of the dimension values of the tuple p. The inverse function $Z^{-1}(.)$ is calculated by reversing the interleaving process and consequently it has the same complexity as *Z (.)*. Thus the Z-ordering scheme is very easy to implement.

Figure 4. Bit interleaving using Z*-order for* 4X4 *universe*

It is to be noted that a one-to-one correspondence of bits in the address to bits in the dimension values allows for algorithms to work only with addresses. An implication of this is that coordinate comparisons can be performed solely on addresses. Bit-interleaving is very useful for on geospatial files because it extends easily to any number of dimensions, and dimensions with differing cardinalities by considering only those dimensions for bit-interleaving that have bits left for interleaving, i.e., if all bits of a dimension are exhausted it will not be considered for the interleaving process anymore. The *Z*-curve and the bit interleaving process are explained in-depth in Foundations of Multidimensional and Metric Data Structures (Samet 2005).

Key-Value stores

Our prototype implementation of GeoBase is done on HBase, which is an open-source key-value store built on top of the Hadoop file system (HDFS). In HBase, keys are stored in byte-lexicographical sorted

order across distributed regions, which range partitions the key space. Its architecture comprises a three layer B+ tree structure; the regions storing data constitute the lowest level. The two upper levels are special regions storing root and meta information. If a region's size exceeds a configurable limit, HBase splits it into two sub-regions. This allows the system to grow dynamically as data is inserted, while dealing with data skew by creating finer grained partitions for hot regions. In its data model, HBase uses the concept of column families, which is a set of columns within a row that are typically accessed together. All the values of a specific column-family are stored sequentially together on the disk. Such a data layout results in fast sequential scans on the consecutive rows, and also on the adjacent columns within a column family in a row.

HBase relies on HDFS for durability. Each key-value pair is multi-versioned and an update to a key-value pair is appends data in the file system. Updates are first written onto memory buffers, and flushed periodically to the disk. As a result writes are faster than reads. HBase can typically handle hundreds of thousands of inserts per second while efficiently processing range queries in real- time with response times as low as hundreds of milliseconds (George 2008). In a vanilla use of key-value stores for multi-dimensional data, *Z*-value of the dimensions being indexed; for instance the location and timestamp become the key and the values correspond to the data values at these indexed positions. As described in the next section, this mapping of keys to *Z*-values imposes significant data translation costs.

GEOBASE

We describe the architecture of GeoBase that enables end-to-end analysis, a few optimization issues that arise in its implementation, and finally describe our query algorithms.

Architecture

The key idea in GeoBase is to cluster regions of space, in- stead of individual data values, to preserve spatial proximity, and interface with the native file system. A region of space corresponds to a space covered by an interval on the *Z*-curve, more commonly called as the *Z*-region (Fenk 2002). The *Z*-region has a linear and spatial interpretation, as shown in the Figure 5, which shows a linear region 'G: [4:20]' and a spatial extent of the *Z*-region. Since the *Z*–region corresponds to the first and last *Z*-values of tuples in a spatial extent; *Z*-regions can themselves be calculated based on bit interleaving, which is the most efficient means to calculate a *Z*-value. In GeoBase indexing based on *Z*-regions preserves spatial proximity though not at a point level, but only at the block level. Thus, neighboring blocks are spatially close with high probability (Fenk 2002).

Figure 5. A Z-region [α:β] is the space covered by an interval of Z-curve and is defined by two Z-addresses α and β. The letters denote a single contiguous region.

To realize this clustering GeoBase implements a data storage layer and an index layer. The data storage layers maps one more file blocks to a physical storage abstraction termed file bucket. The overlaid index layer consists of key-value pairs, in which each key corresponds to a region of space, and value maps to blocks in the file bucket. Since the key- value pairs are sorted, the index imposes an ordering over the region space. The mapping between a region in the index layer and a file bucket in the data storage layer can be one-to-one, many-to- one, or many-to-many depending on the implementation and requirements of the application.

Figure 6. GeoBase Architecture. The letters denote Z–regions

Figure 6 shows the interaction between index layer and storage layer. In this figure, space is partitioned into 4 disjoint *Z*-regions. The region marked 'G' and 'P' consists of two disconnected parts, but as we see the index layers maintains the spatial proximity of the two regions by mapping neigh- boring *Z*-regions to file buckets and keeping them together. In addition, for *Z*-ordering regardless of the dimensionality of the *Z*-ordered space (i.e., not only for 2d) the number of not connected parts of a *Z*-region is at most two. It is to be noted that HBase internally stores key-value pairs as a B+tree and *Z*-regions can be stored in a B+tree by storing the upper limit of the *Z*-value.

4.2.2 Optimization Issues

Partitioning: The data storage layer is built over HDFS, which offers default partitioning. However, the default partitioning ignores file layout knowledge encoded in NetCDF files, and thus effectively disables any automatic optimizations that rely on such knowledge. We know that a NetCDF file is divided into metadata and data sections as described in Section 4.1.1. In addition, the metadata may occur at any place in the file. This occurrence of metadata often does not lead to even partitioning of the multi-dimensional space into blocks. Thus some blocks will have more data values and some more metadata. In addition a single time dimension may get split across blocks.

We adopt three optimizations with respect to partitioning a NetCDF file. In the first optimization, file data is partitioned into blocks based on logical array data model. Thus, a NetCDF file with a variable v is partitioned into n blocks, each equal in the size of the array. Practically, this scheme translates to unequal data distribution because of presence of metadata. In the second optimization, we partition the file based on the actual distribution of data and metadata. So for each file we calculate the bytes occupied by the meta- data and data and partition each block into equal byte sized regions. Finally, our last partitioning scheme is to split files such that all metadata is copied to one block, and data is split based on the logical level. This scheme requires some data rearranging and processing prior to reading the input splits.

Load Balancing: Some load balancing is built into HBase as HDFS distributes the blocks randomly among data nodes. In addition the index layer is B+tree, which is also balanced. However, since HBase follows a range-partitioning scheme on the key-space, and, given skew in the data, certain regions become hot while querying. One alternative is to use other key-value stores, such as MongoDB and Voldemort, which support a hash-partitioning scheme, but would require implementing another index layer to support simple range queries.

To improve load balancing, a simple optimization heuristic is that a node attempts to shed its load whenever its load increases by a factor δ, and attempts to gain load when it drops by the same factor. In GeoBase this corresponds to moving file buckets from one physical node to another. But since HDFS provides some level of replication, in practice no data needs to be moved, only the index layer randomly chooses from the many mappings of the index element to the file buckets. This strategy as we show in experiments increases the response time of query, but not in a significant way.

GeoBase can be used to answer point queries and range queries. Determining the *Z*-region to which a point belongs forms the basis for point queries. Since the index layer comprises a sorted list of *Z*-region names, determining the region to which a point belongs is efficient. Recall that the region name determines the bounds of the region. To answer a point query, we first lookup the *Z*-region that corresponds to the *Z*-value of the point. The point is then queried in the bucket to which the *Z*-region maps.

In the range query, first calculate the start and the end *Z*-values of the query box coordinates q_l and q_h. We calculate the set of candidate *Z*-regions that will satisfy the range. All those regions within the query box

are candidate regions, which can be pipelined for further processing. Note that any false positives are eliminated by a scan of the index. As the region name only is enough to determine the boundary of the region enclosed by the subspace, points in a region are scanned only if the range of the subspace intersects with the query range.

Nearest neighbor queries are also an important primitive operation for geoscience applications, but are currently a work in progress in GeoBase. The index layer is a sorted sequence of subspace names. Since the subspace names encode the boundaries, the index layer essentially imposes an order between the subspaces. In our prior discussion, we represented the index layer as a monolithic layer of sorted subspace names. A single partition index is enough for many application needs. Assume each bucket can hold about 10^6 data points. Each row in the index layer stores very little information: the subspace name, the mapping to the corresponding bucket, and some additional metadata for query processing. Therefore, 100 bytes per index table row is a good estimate. The underlying key-value store partitions its data across multiple partitions. Considering the typical size of a partition in key-value stores is about 100 MB, the index partition can maintain a mapping of about 10^6 subspaces. This translates to about 10^{12} data points using a single index partition. This estimate might vary depending on the implementation or the configurations used. But, it provides a reasonable estimate of the size.

The data storage layer is implemented in which we allocate a table per bucket. The data storage layer therefore consists of multiple HBase tables. This model provides flexibility in mapping subspaces to buckets and allows greater parallelism by allowing operations on different tables to be dispatched in parallel.

Partitioning and placement of data is implemented using low-level customizations to HDFS default partitioning and placement that targets text-based formats. Hadoop's *FileInputFormat* class, performs chunking and grouping which result in a set of partitions, each related to a host on which the partition should be processed. A partition in the *InputSplit* class represents a subset of the total logical input space. Each *InputSplit* and associated partition is placed transparently by HDFS using higher-level policies such as load balancing. The data storage layer indexes via the *InputSplit* class by reading the associated logical space from the underlying file system. The reading capability is built into Hadoop's RecordReader class that utilizes the NetCDF-Java library to access data specified at the logical level.

In order to support logical-to-physical translation, we extended the NetCDF-Java library to expose mapping information. The translation mechanism is exposed via a function that takes as input a set of logical coordinates, and produces a set of physical byte stream offsets corresponding to each coordinate. Internally this is implemented as a simulation of the actual request that NetCDF would perform, but does not complete the read to the underlying file system, and instead responds with the offset of the request.

Additional software layers also had to be created to allow NetCDF-Java to interoperate with HDFS. NetCDF-Java is tightly integrated with the standard POSIX interface to reading files. Unfortunately, HDFS (the distributed file system built for Hadoop), does not expose its interface via standard POSIX system calls. To accommodate this API mismatch we built an HDFS connector, which translated POSIX calls issued by NetCDF-Java into calls to the HDFS library.

EXPERIMENTS

We conducted an experimental study to measure the performance of GeoBase on representative queries and data.

The setup consists of a cluster ranging from 10 to 15 small instance nodes, provisioned on Amazon EC2. The cluster is homogeneous in that each node consists of 1 virtual core, 1.7GB memory, 160 GB HDD, and 64bit Ubuntu 10.10. The deployed version of Hadoop is 0.22 and HBase 0.20.6. The Hadoop cluster is configured with 1 master node and the remaining HDFS and HBase nodes.

Methodology

Our datasets consist of 24 weather variables that are of high importance to crop models and coordinate and time attributes. There are 27,000 files in the system. Each files stores daily values of each variable over a grid that spans a land mass and the water region. Our queries are generated synthetically. Each query is either a data reduction query or a data condensation query. A data reduction query selects 5-40 distinct regions from the grid, and a data condensation query selects from computing on a max, min or median. A total of 10,000 queries are executed over a period of two hours with varying arrival times. The query workload that specifies the weather variables and the set of ranges is generated synthetically, but closely corresponds to queries that come for data over standard crop model simulations.

The primary end-to-end metrics used in evaluating GeoBase is the time to make an incoming file interface with the analysis system and then the runtime of the query. In addition, we also look into throughput, CPU utilization, and HDFS Local Reads. The latter is a measurement of the fraction of the input to the range query that was read from a local disk managed by HDFS.

Results

In Section 3, we demonstrated the performance of an end- to-end system with access enabled through data access libraries (**DA**). We also showed performance results when large amounts of data in NetCDF format is first transformed into text format and then accessed through 1-dimensional ordering, where the one dimension is one of the spatial coordinates (**1D**). We consider two other baseline end-to-end implementations: First, transforming every incoming file's input value into z-order and then inserting the *Z*-order values with using z-ordering for linearization (**ZO**) without any specialized index. Our other baseline implementation uses the plugin for NetCDF in Hadoop to run MapReduce style data reduction and condensation queries. Thus the queries are implemented in Map Reduce to evaluate the performance of a MapReduce system (**MR**) performing a full parallel scan of the data. Finally, we run the queries over the GeoBase system (GB).

Figure 7. All end-to-end systems compared (on log scale)

To conduct the experiment we inserted 27000 files, in batches of multiples of 1000 files into the data ingestion system, and then queries corresponding to the time inserted so far were run on the ingested files. On the y-axis we report the total end-to-end time for the data ingestion and query execution (Figure 7). The experiment over increasing number of cluster nodes shows that the performance of GB scales linearly and continues to outperform **ZO** and **MR**, with **DA** and **1D** being poor choices. In fact, **MR** performs better than the **ZO** showing that the cost of translating every data value into **ZO** is very large. But then full scans in **MR** are not particular useful. The average query selectivity is 50% over the entire dataset so most nodes conduct the scan but lead to low actual yields for the reduction operator. In GeoBase (GB) files do not incur any data transformation costs but are available immediately. The Z-region is indexed within the indexing layer with very low overhead, and each inserted file is incrementally indexed with some processing overhead.

Query performance

We now evaluate range query performance using the different implementations of the index structures and the storage layer and compare performance with **ZO** and **MR**. The **MR** system filters out points matching the queried range and reports aggregate statistics on the matched data points. In GeoBase we choose the no-load-balancing scheme and file data is partitioned based on the first scheme in which data and second scheme in which a file data and metadata is partitioned based on logical model and the number of bytes in each partition, respectively. We then choose load balancing with second partitioning scheme.

Figure 8. Response times for range query as function of selectivity.

Figure 8 plots the range query response times for the different designs as a function of the varying selectivity. As is evident from Figure 8, GeoBase with its design choices outperform the baseline systems. In particular, for highly selective queries where our designs show a one to two orders of magnitude improvement over the baseline implementations using simple **ZO** or **MR**. Moreover, the query response time of our proposed designs is proportional to the selectivity, which asserts the gains from the index layer when compared to brute force parallel scan of the entire data set as in the **MR** implementation whose response times are the worst amongst the alternatives considered and is independent of the query selectivity. In the design using just *Z*-ordering (**ZO**), the system scans between the minimum and the maximum *Z*-value of the queried region and filters out points that do not intersect the queried ranges. If the scanned range spans several buckets, the system parallelizes the scans per bucket. **ZO** response time is almost ten times worse when compared to our proposed approaches, especially for queries with high selectivity. The main reason for the inferior performance of the **ZO** is the false positive scans resulting from the distortion in the mapping of the points introduced by linearization.

File insert throughput

We describe the performance of inserts of file batches into the GeoBase. Figure 9 shows the near-constant performance over increasing number of files, which demonstrates the scalability of the files. The performance is due to the fact that GeoBase incurs no data transformation costs while inserting files. It is to be noted that when a file is inserted it is simply written onto the file system. The insertion of the file notifies the storage layer, and the corresponding *Z*- regions on the file are computed, which are then indexed by the index layer. Sometimes, a file bucket has to split or an index has to split. Thus we see some cost increasing with the number of files, but it is not hugely significant. Also the system block other operations until the bucket split completes. In our experiments, this required about 30 to 40 seconds to split a bucket.

Figure 9. Throughput of file inserts

Optimizations

We consider file partitioning optimization and load balancing. File partitioning optimization has significant impact on the end-to-end performance of GeoBase. Figure 10 shows the end-to-end time for the three schemes as described in Section 4.2.2. While the third partitioning scheme performs the best it also has an additional data transformation cost, which is 1.5x more than the cost incurred by other schemes. Though, the scheme achieves the best data locality, achieving close to 90% local reads. The other schemes also show good data locality with experiments going from 9% local reads to 80% local.

Figure 10. Partitioning Schemes compared

HBase employs automatic and dynamic load balancing scheme onto the region to scale the system. For instance, if each region is configured to account for 100MB of data, the region will split into 2 regions

with each of them accounts for 50MB of data when this threshold (i.e. 100MB) is exceeded. We instrumented the dynamic load-balancing scheme in HBase to conduct region splits when the threshold exceeds δ times, in which delta is varied from 1 to 2. We used a synthetic spatially skewed data set using a Zipfian distribution with a Zipfian factor of 1.0 representing moderately skewed data. Using a Zipfian distribution allows us to control the skew while allowing quick generation of large data sets. Figure 11 shows when load is moved and GeoBase performance under the variation. The system is compared with **ZO** and **MR** and GeoBase has better performance under all load variations.

Figure 11. Load Balancing

FUTURE RESEARCH DIRECTIONS

Scientists are increasingly facing big data issues but are not equipped with appropriate tools to manage it. In the case of geosciences, large amounts of data are now available, but its analysis is limited by non-scalable access methods that affect analysis and visualization methods. There is an emerging need to reduce scientists' data management tasks, especially scientists' directing single-investigator laboratories, comprising of a few graduate students and technicians, and yet striving to make their data available to other community members for sharing. Such scientists need end-to-end systems and services that are efficient, robust, and scalable. GeoBase, is a step in that direction, interfacing directly with files and providing a scalable querying service.

Our immediate future research directions is to explore such an end-to-end approach for log-structured merge trees (O'Neil 1996), which will be more suitable for rapidly streaming multidimensional data that arises from simulation experiments.

CONCLUSION

In a data-centric approach, scientists typically first explore data using standard statistical analysis tools and visualization, and then perform sophisticated data mining to extract deeper information and create knowledge. Given ever-increasing data volumes, scientists often find themselves conducting more data storage and management tasks rather than entirely focusing on data exploration and data mining. Most providers of large datasets provide simple mechanisms for data exploration, such as search based on metadata. Consequently, scientists have to first download large volumes of data, learn about storage management and data organization, and do so in a scalable way given the sheer size of data, before undertaking any meaningful data exploration and data mining.

In this chapter, we have presented GeoBase, a parallel data access system for scalable analysis that is based on open-source HBase. A vanilla installation of HBase is not sufficient for Geoscience applications since it neither provides interface to commonly used self-describing formats nor multi-attribute access, essential to for accessing data from spatio-temporal datasets. GeoBase provides an end-to-end solution by natively incorporating support for both.

REFERENCES

Abadi, D., Madden, S., & Hachem, N. (2008). Column-Stores vs. Row-Stores: How Different Are They Really? 2008 ACM SIGMOD International Conference on Management of Data.
Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., & Rasin, A. (2009). "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads." Proceedings of the VLDB Endowment **2**(1): 922-933.

Apache. (2010). "HBase: Bigtable-like Structured Storage for Hadoop HDFS." from http://hadoop.apache.org/hbase/.
Baumann, P. (1999). A Database Array Algebra for Spatio-Temporal Data and Beyond. . Workshop on Next Generation Information Technologies and Systems.
Buck, J., Watkins, N., LeFevre, J., Ioannidou, K., Maltzahn, C., Polyzotis, N., & Brandt, S. (2011). SciHadoop: Array-based Query Processing in Hadoop. Supercomputing.
Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. (2008). "BigTable: A Distributed Storage System for Structured Data." ACM Transactions on Computer Systems (TOCS) **26**(2).
Fenk, R., Markl, V., & Bayer, R. (2002). Interval Processing with the UB-tree. Database Engineering and Applications Symposium.
Folk, M., Cheng, A., & Yates, K. (1999). HDF5: A File Format and I/O Library for High Performance Computing Applications. Supercomputing.
GDAL Geospatial Data Abstraction Library.
George, L. (2008). HBase: The Definitive Guide, O'Reilly Media.
Hill, D., & Werpy, J. (2011). Satellite Imagery Production and Processing Using Apache Hadoop. American Geophysical Union Fall Meeting Abstracts.
Howe, B. (2007). GridFields: Model-Driven Data Transformation in the Physical Sciences. PhD, Portland State University, Portland OR.
Jagadish, H. (1997). "Analysis of Hilbert Curve for Representing 2-Dimensional Space." Information Processing Letters **62**(1): 17-22.
Jensen, C., Lin, D., & Ooi, B. (2004). Query and update efficient B+-tree based indexing of moving objects. Proceedings of the Thirtieth International Conference on Very Large Data Bases.
Li, J., Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., & Zingale, M. (2003). Parallel NetCDF: A High Performance Scientific I/O Interface. Supercomputing.
Libkin, L., Machlin, R., & Wong, L. (1996). "A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques." ACM SIGMOD Record **25**(2): 228-239.
Lu, M., & Zwaenepoel, W. (2010). HadoopToSQL: A MapReduce Query Optimizer. European Conference on Computer Systems.
Marathe, A., & Salem, K. (1997). A Language for Manipulating Arrays. Very Large Databases.
NASA. "AQUA." from http://aqua.nasa.gov/.
NASA. "Landsat 7." from http://landsat.gsfc.nasa.gov.
NASA. "Pomegranate." from http://pomegranate.nasa.gov/.
Nishimura, S., Das, S., Agrawal, D., & Abbadi, A. (2011). MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. 12th IEEE International Conference on Mobile Data Management.
NOAA. "Geostationary Satellite Server." from http://www.goes.noaa.gov/.
O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). "The Log-structured Merge-Tree (LSM-tree)." Acta Informatica **33**(4).
OPeNDAP, I. "OPeNDAP." from http://opendap.org/.
Rew, R., & Davis, G. (1990). "NetCDF: An Interface for Scientific Data Access." IEEE Computer Graphics and Applications **10**(4).

Samet, H. (2005). Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc.
Stonebraker, M., Becla, J., Dewitt, D., Lim, K., Maier, D., Ratzesberger, O., & Zdonik, S. (2009). Requirements for Science Databases and SciDB. Conference on Innovative Database Research.
Tao, Y., Yi, K., Sheng, C., & Kalnis, P. (2009). Quality and Efficiency in High Dimensional Nearest Neighbor Search. . Proceedings of the 35th SIGMOD International Conference on Management of Data.
Unidata. "NetCDF Java API." from http: //http://www.unidata.ucar.edu/software/netcdf-java/.
Wang, D., Zender, C., & Jenks, S. (2008). Clustered Workflow Execution of Retargeted Data Analysis Scripts. 8th IEEE International Symposium on Cluster Computing and the Grid.
Wu, K., Ahern, S., Bethel, E., Chen, J., Childs, H., Cormier-Michel, E., Geddes, C., Gu, J., Hagen, H., Hamann, B., & others (2009). "FastBit: Interactively Searching Massive Data." Journal of Physics **180**(2).
Zhao, H., Ai, S., Lv, Z., & Li, B. (2010). Parallel Accessing Massive NetCDF Data Based on MapReduce. . Web Information Systems and Mining, , Springer Lecture Notes in Computer Science.

ADDITIONAL READINGS

Egenhofer, M. J. (1994). Spatial SQL: A query and presentation language. *Knowledge and Data Engineering, IEEE Transactions on*, *6*(1), 86-95.
George, L. (2011). HBase: The definitive guide. O'Reilly Media, Inc.
Hjaltason, G. R., & Samet, H. (2003). Index-driven similarity search in metric spaces (Survey Article). *ACM Transactions on Database Systems (TODS)*, *28*(4), 517-580.
Lam, C. (2010). *Hadoop in action*. Manning Publications Co..
Lin, J., & Dyer, C. (2010). Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, *3*(1), 1-177.
Miller, M. (2008). *Cloud computing: Web-based applications that change the way you work and collaborate online*. Que publishing.
Rew, R., Davis, G., Emmerson, S., Davies, H., & Hartnett, E. (1993). *NetCDF User's Guide*. Unidata Program Center, Boulder, Colorado.
Samet, H. (1990). *The design and analysis of spatial data structures* (Vol. 199). Reading, MA: Addison-Wesley.
Silberschatz, A., Korth, H. F., & Sudarshan, S. (1997). *Database system concepts* (Vol. 4). Hightstown: McGraw-Hill.
Vora, M. N. (2011, December). Hadoop-HBase for large-scale data. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on* (Vol. 1, pp. 601-605). IEEE.
White, T. (2012). *Hadoop: the definitive guide*. O'Reilly.

KEYTERMS

Data-intensive computing is a class of parallel computing applications which use a data parallel approach to processing large volumes of data typically terabytes or petabytes in size and typically referred to as Big Data.

Multidimensional structure is defined as “a variation of the relational model that uses multidimensional structures to organize data and express the relationships between data”

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster.

A **NoSQL** database provides a mechanism for storage and retrieval of data that uses looser consistency models rather than traditional relational databases.

A **space-filling curve** is a curve whose range contains the entire 2-dimensional unit square (or more generally an n-dimensional hypercube).

Z-order, **Morton order**, or **Morton code** is a function, which maps multidimensional data to one dimension while preserving locality of the data points.

A **file format** is a standard way that information is encoded for storage in a computer file.

NetCDF (**Network Common Data Form**) is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.

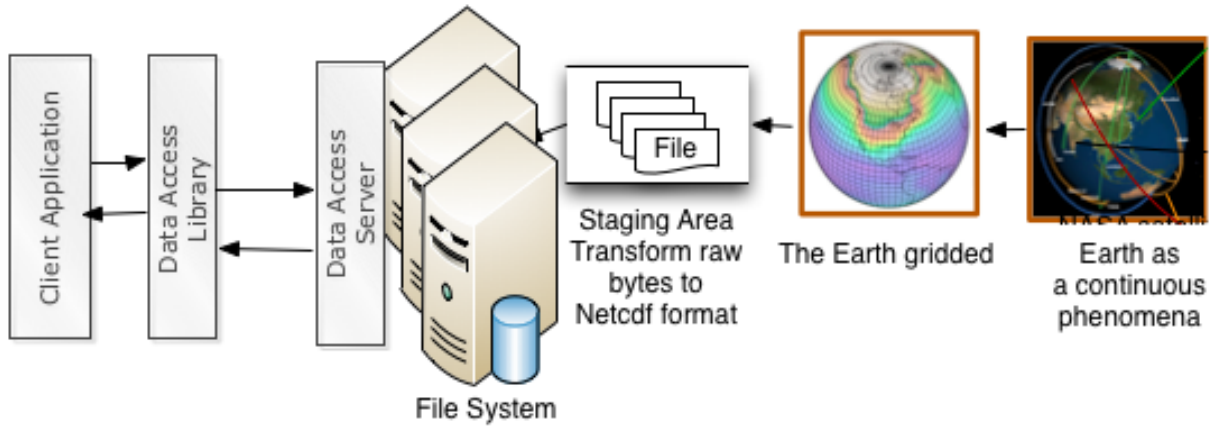

Figure 1. ***Measurement of sea surface temperature and pressure variables on Earth. The continuous phenomenon is discretized into a regular grid; each grid cell records the values of the physical variables under its spatial extent, and is manifested into a file, based on a chosen format, such as NetCDF, HDF5, GRIB.***

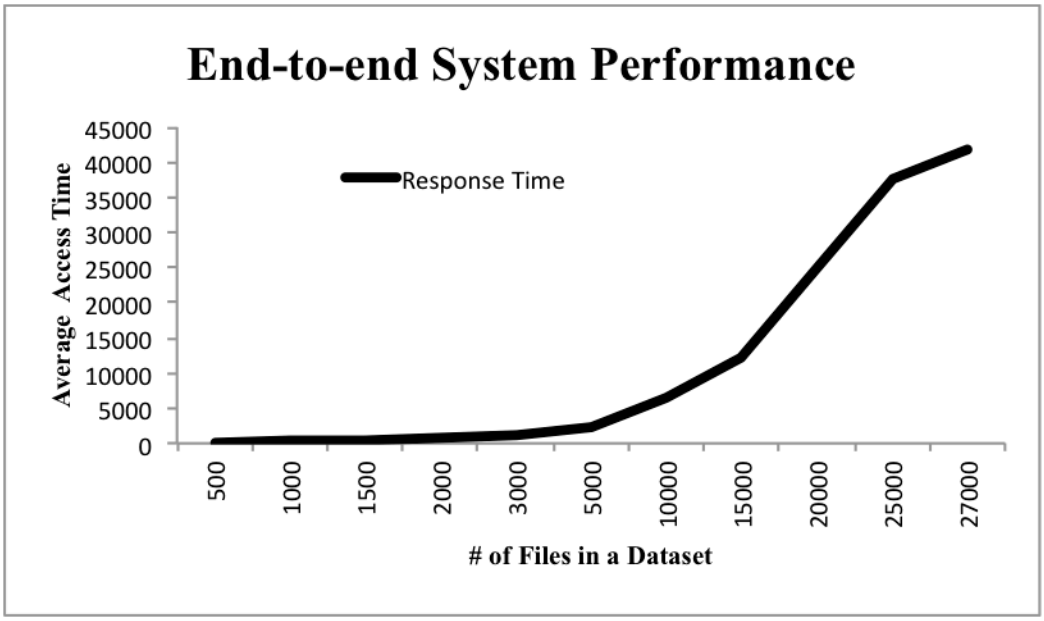

Figure 2. **End-to-end system performance of data ingestion system**

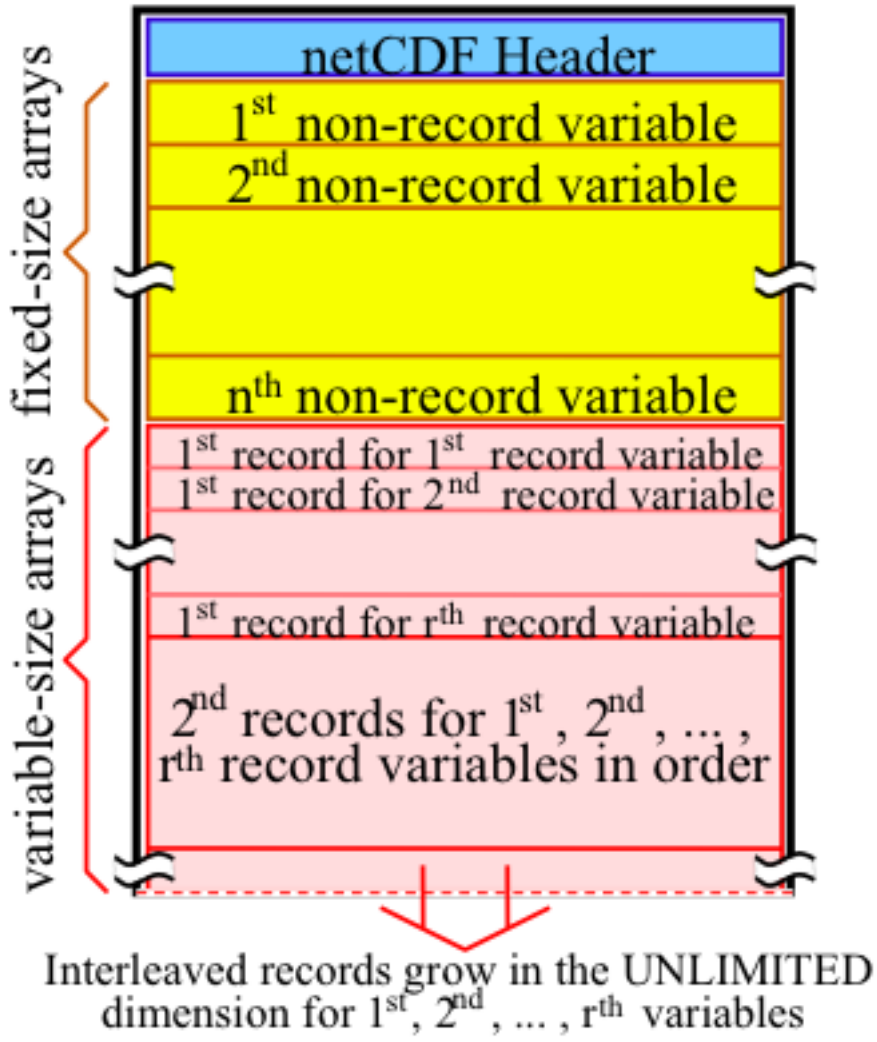

Figure 3. NetCDF File Format. (Source: Li, Liao, Choudhary, et. al (Li 2003))

11	0101	0111	1101	1111
10	0100	0110	1100	1110
01	0001	0011	1001	1011
00	0000	0010	1000	1010
	00	01	10	11

Figure 4. Bit interleaving using Z-order for 4X4 universe

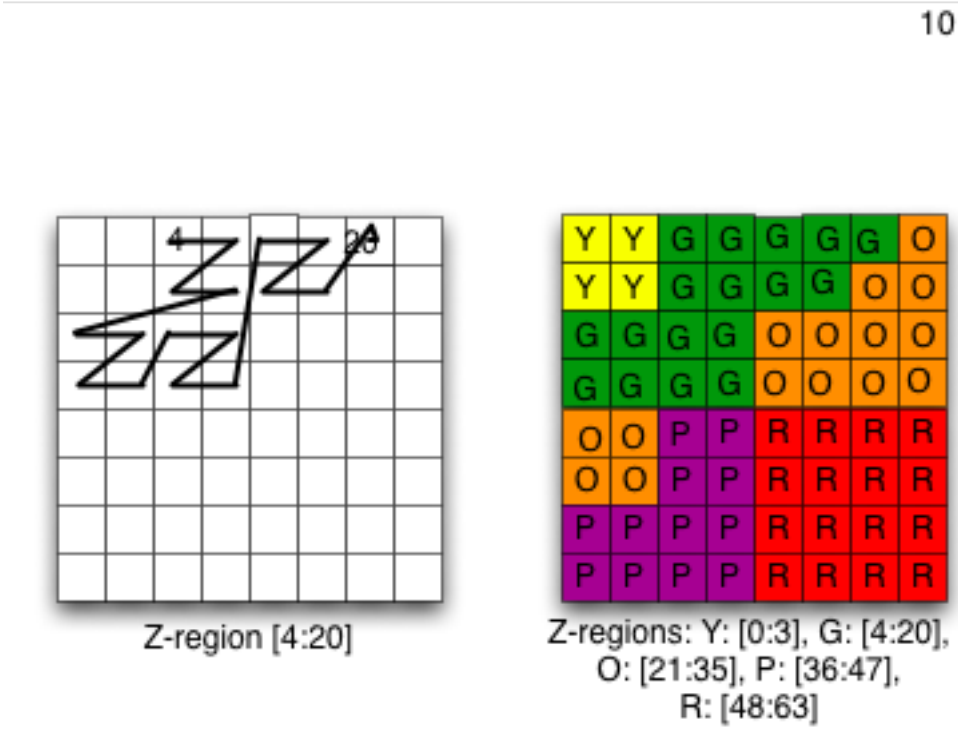

Figure 5: A Z-region [α:β] is the space covered by an interval of Z-curve and is defined by two Z-addresses α and β. The letters denote a single contiguous region.

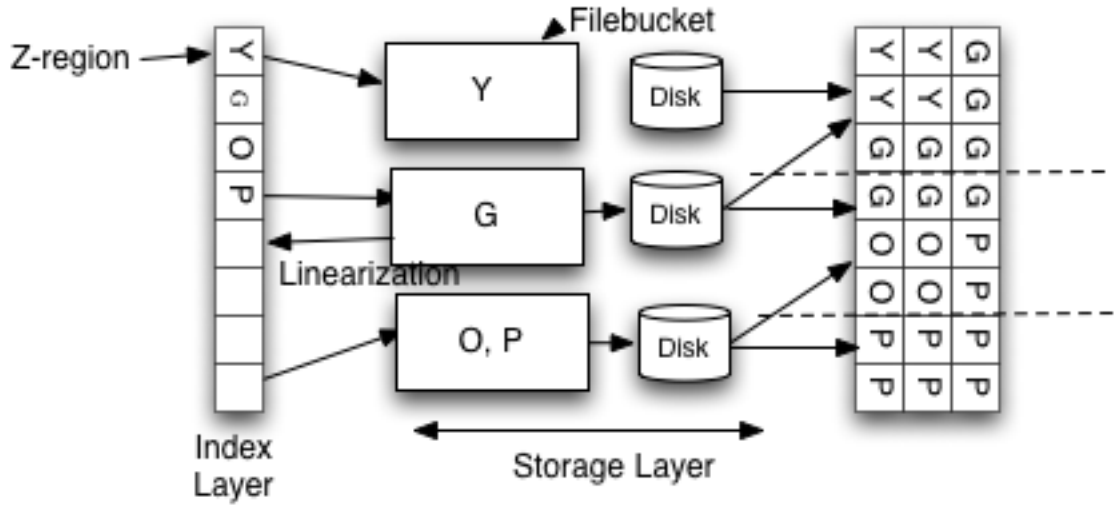

Figure 6. ***GeoBase Architecture. The letters denote Z–regions***

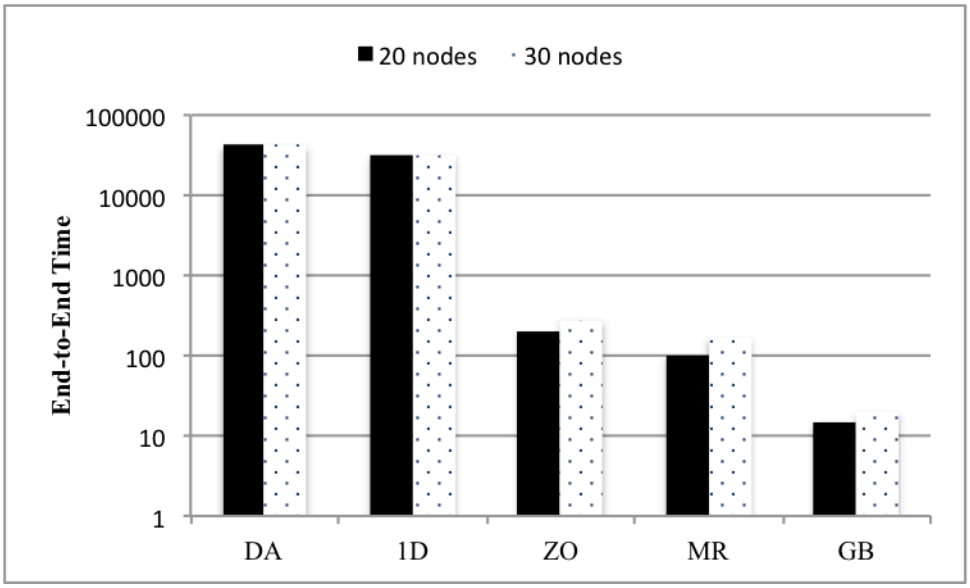

Figure 7. All end-to-end systems compared (on log scale)

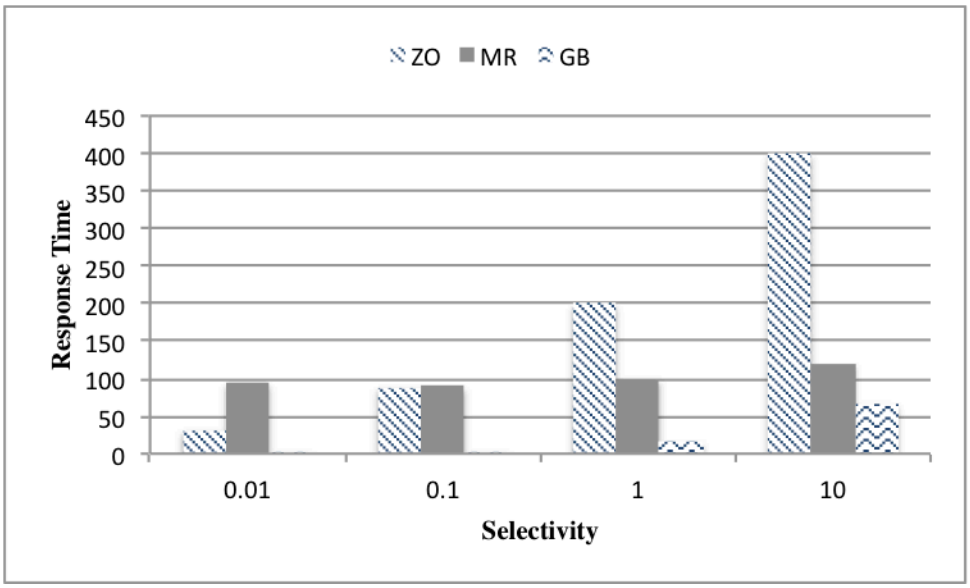

Figure 8. Response times for range query as function of selectivity.

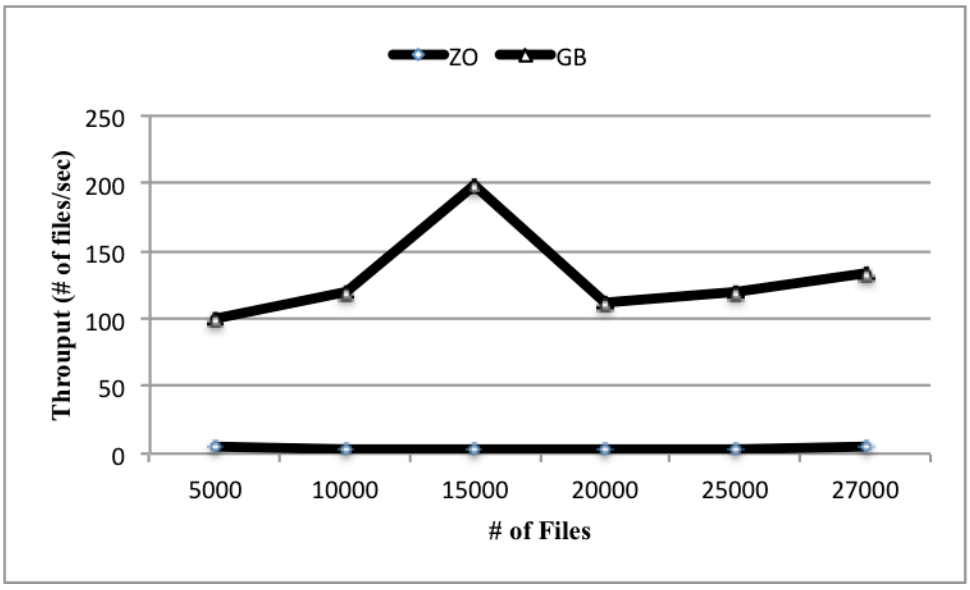

Figure 9 Throughput of file inserts

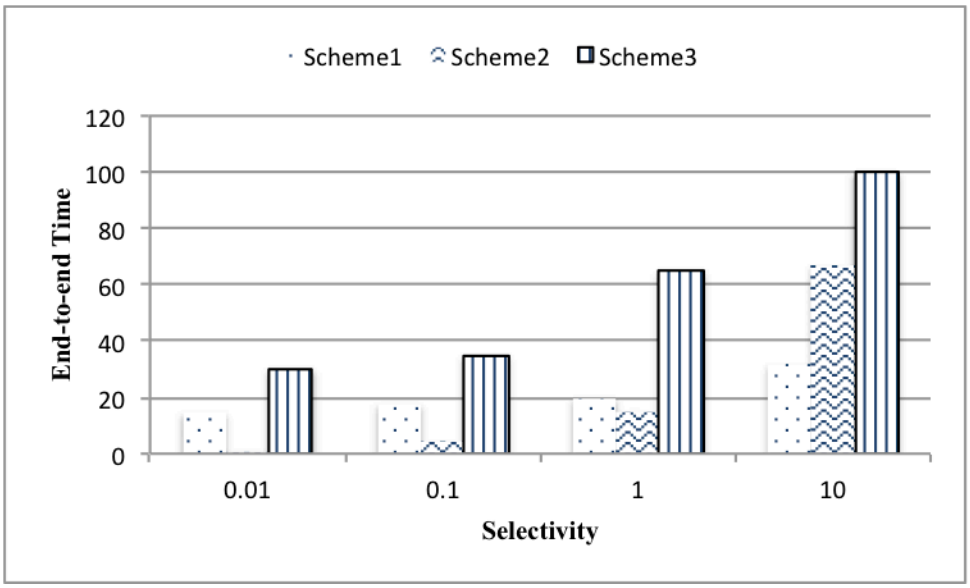

Figure 10. Partitioning Schemes compared

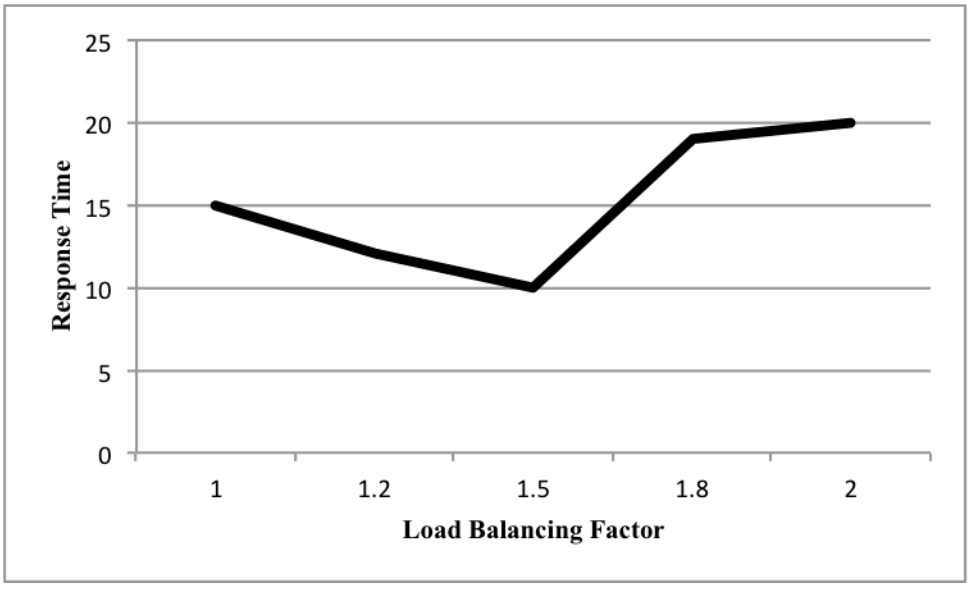

Figure 11. Load Balancing