Griddle: Effective Query Support over Voluminous Gridded Spatial Datasets

Pierce Smith
Department of Computer Science
Colorado State University
psmith36@rams.colostate.edu

Sangmi Lee Pallickara
Department of Computer Science
Colorado State University
sangmi@colostate.edu

Shrideep Pallickara
Department of Computer Science
Colorado State University
shrideep@cs.colostate.edu

***Abstract*—Gridded datasets occur in several domains. These datasets comprise (un)structured grid points, where each grid point is characterized by XY(Z) coordinates in a spatial referencing system. The data available at individual grid points are high-dimensional encapsulating multiple variables of interest. This study has two thrusts. The first targets supporting effective management of voluminous gridded datasets while reconciling challenges relating to colocation and dispersion. The second thrust is to support sliding (temporal) window queries over the gridded dataset. Such queries involve sliding a temporal window over the data to identify spatial locations and chronological time points where the specified predicate evaluates to true. Our methodology includes support for a space-efficient data structure for organizing information within the data, query decomposition based on dyadic intervals, support for temporal anchoring, query transformations, and effective evaluation of query predicates. Our empirical benchmarks are conducted on representative voluminous high dimensional datasets such as gridMET (historical meteorological data) and MACA (future climate datasets based on the RCP 8.5 greenhouse gas trajectory). These datasets represent excellent proving grounds to validate several aspects of our methodology. We show that in a clustered environment, our system can handle throughputs of over 3000 mutli-predicate sliding window queries per second, summarizing over 1 TB of data in well under 10 GB of memory on each individual machine.**

***Index Terms*—gridded high-dimensional datasets, multi-predicate spatiotemporal queries, voluminous data collections, distributed indexing structures**

I. Introduction

Gridded datasets are made available in several domains and play a key role in modeling spatial phenomena. The grid points in these datasets represent (latitude, longitude) coordinates that uniquely identify a spatial location. At each grid point a vector of variables representing features of interest are made available. Gridded datasets are characterized by their resolutions, spatial extent under consideration, the chronological time range, and the recorded variables. The resolution corresponds to the spatial increments at which data are made available e.g., 1 km, 4 km, etc. The spatiotemporal scope represents the spatial extent and the chronological time range of the data. The frequency corresponds to the timesteps at which the data are available for each grid point.

Gridded datasets represent outputs from modeling and simulations. In several cases, these modelling and simulations typically also take observational data from targeted locations. In some cases, actual historical observations are combined with statistical smoothing and interpolation techniques to produce gridded datasets that are amenable to fine-scale analysis. An exemplar of such a dataset is the gridMET dataset which provides historical meteorological datasets. These datasets provide a wealth of information and offer opportunities to combine analysis with disparate datasets to inform decision-making.

Gridded datasets are important because they support exploration, hypothesis formulation, and analysis at diverse spatiotemporal scales. They are often supplemented with other datasets (gridded, point clouds, or shape files) from other domains to explore/understand domain-specific phenomena. For example, an urban planning researcher may combine analysis that incorporates data from Census surveys, future climate gridded datasets, and infrastructure data from HIFLD to identify vulnerabilities to heat exposure. Another example involves combining gridded climate datasets with soils and evaluation information to identify locations that are vulnerable to pooling and flooding.

Our objective is to effectively support sliding window queries over voluminous gridded datasets. A sliding window query is characterized by the temporal bookends, the spatial constraint, the query predicate, and the timesteps. The objective is to evaluate the query predicate by sliding the window across time. Results from the query identify spatial locations (or administrative boundaries) that satisfy the query predicate for every time step encapsulated within the sliding window. This entails sliding a temporal window over the spatially constrained dataset and testing to see if the constraints are satisfied within that window. Across different, concurrent queries that are evaluated, the size of the temporal window, the spatiotemporal scope, the variables involved, and the specified query predicates could all be different.

Our sliding window queries identify spatial extents and the time-periods where the query predicates evaluate to true. Traditional indexing schemes may not work well for such queries. The deficiency stems from the spatiotemporal components associated with the multivariate data alongside the data volumes. The nature of query evaluations in gridded datasets involves maintaining cumulative density functions over multivariate data; indexing stores that are optimized for point query evaluations struggle to support this out of the box.

While SQL or NoSQL stores could be used to construct these queries, the number of data sweeps and the disk I/O that will likely be incurred can result in prolonged query evaluation latencies.

A. *Challenges*

Supporting sliding window queries over spatiotemporal gridded datasets introduces several challenges:

1) Volume: The datasets we consider are voluminous (Peta-scale) and individual observations comprising the data collections are high dimensional. The billions of observations comprising the dataset are encoded as vectors.
2) Spatiotemporal scopes: The observations are dispersed over large spatiotemporal scopes. For example, the datasets we consider cover the entire contiguous United States and include multidecadal data. Further, the data often needs to be collated across administrative or political boundaries.
3) Disk I/O: The speed differential across the memory hierarchy exacerbates challenges. The disk I/O subsystem is about six orders of magnitude slower than main memory, but the dataset sizes far exceed the memory capacity available within distributed clusters.
4) Query semantics: Spatiotemporal queries incorporate spatial (geometry) and chronological (time) constraints in addition to the typical predicate logic. Given the parameter space evolution and multivariate relationships across different spatiotemporal scopes, care must be taken to ensure efficiency of queries. Methods that rely on spatiotemporally-agnostic indexing schemes may end up necessitating multiple, repeated sweeps of the data that adversely impact both the latency and throughput of query evaluations.
5) Sliding window queries: Queries may specify an arbitrary sized temporal window over which the query predicates must hold true. The results of such a query evaluation represents a "sliding" of the window over the entire chronological extent encapsulated by that dataset. Window-agnostic indexing schemes may trigger a sweep of the entire dataset for every query. The system should be able to support multiple concurrent query evaluations each of which may encapsulate diverse spatial extents, temporal window sizes, and query predicates.

B. *Research Questions*

Research questions that we explore include:

RQ-1: *How can we effectively stage and manage voluminous datasets?* In particular, we wish to support data colocations based on spatial extents while supporting aggregation along administrative boundaries.

RQ-2: *How can we design data structures to effectively index such data?* These data structures must be amenable to both incremental and batched updates.

RQ-3: *How can we incorporate support for queries and refinements to facilitate timely evaluations at scale?* Concurrent query evaluations must be supported to ensure that diverse sliding window queries may be evaluated at high throughput.

C. *Approach Summary*

To support low-latency sliding window queries, we construct compact representations of the data. These represent an index in the traditional sense of data stores, i.e., they sit on top of the data and are used to speed up queries. Another way to view this representation is as a sketch that serves as a surrogate for the underlying data when answering queries.

GeoSieve Data Structure and Construction Our data structure, GeoSieve, collects information from individual observations and maintains a space–efficient summary of the dataset as viewed through a fixed-size sliding window.

Construction of a GeoSieve instance is flexible. GeoSieve instances are created per-dataset and can be constructed for any number of variables or sliding window intervals. Instances can be built from any dataset where sliding-window queries over the variables of interest are possible, as initialization requires only a list of observations for each variable that slide over the chronological scope and with the desired temporal granularity.

The structure itself summarizes the data through temporal CDF (Cumulative Density Functions) stored as run-length encoded bit vectors. For each variable and administrative region in the dataset, a set of bit vectors summarize the temporal distribution of the variable, which are organized in a tree-like structure for efficient lookup and traversal during query evaluations.

Dyadic Intervals A single GeoSieve instance summarizes the dataset as viewed through a fixed sliding window size. To support queries of arbitrary sliding window sizes without needing to construct a unique instance for *every* conceivable size, we maintain indices at dyadic intervals; we split and chain queries among multiple dyadic intervals to support arbitrarily long windows.

GeoSieve$\langle 2^T \rangle$ represents the sketch constructed over a temporal window of size 2^T. For example, GeoSieve$\langle 2^3 \rangle$ can answer queries about what happened over a 8-day period (and *only* an 8-day period). Every GeoSieve$\langle 2^T \rangle$ instance supports two query modes: an anchorless mode that retrieves all time windows where the predicate evaluates to true, and an anchored mode where the query includes an anchor (or "starting point") that allows evaluations to be constrained for a 2^T-day window starting at the specified anchor point. The anchored mode allows for multiple queries over different window sizes to be effectively chained together.

Queries are broken down into sub-queries based on dyadic intervals. The query is evaluated first over the largest dyadic interval, and whatever matches are found are used as the anchor for a query over the next largest interval, until all intervals are queried. For example, consider the case where the query window is specified to be 13 days. This is broken up into dyadic intervals of 8,4, and 1. The query is first evaluated against a GeoSieve$\langle 2^3 \rangle$ instance to identify spatiotemporal scopes where the query predicate evaluates to `true` for a window of 8 days. Next, the query is presented to a GeoSieve$\langle 2^2 \rangle$ instance, with spatial and temporal anchors at the results of the previous query, providing spatiotemporal scopes where the

query predicates are `true` for a temporal window of 8 + 4 days. Finally, it is presented to a GeoSieve$\langle 2^0 \rangle$ instance, again with spatial and temporal anchors at the results of the previous query, providing spatiotemporal scopes where the predicates are `true` for a full 13 day interval. To support queries of up to size w, we need only construct $\lceil \log_2(w) \rceil$ GeoSieve instances.

The data structure also supports entirely concurrent evaluation, where the query is raised against each GeoSieve instance in parallel and resulting spatiotemporal scopes are intersected together to calculate the result set. This, however, results in a large amount of discarded information and a less efficient execution, as many spatiotemporal locations that have no chance of being relevant to the original query will be examined.

Ensuring High Throughput The data structure is space–efficient since we maintain the temporal CDFs in a tree comprising run length encoded bit vectors. This allows the entire data structure to be entirely memory resident, even when built over datasets on the order of several terabytes large. This memory residency avoids costly disk and network access that would be required of traditional sliding window queries to most databases, while ensuring high throughput and low latency.

The GeoSieve data structure is thread-safe, allowing for concurrent reading. No shared, mutable state is involved in traversing the data structure; thus, multiple threads may evaluate concurrent, unrelated queries over the same data. For queries that comprise multiple predicates, several threads may also work on the same query at the same time. Queries only need to be blocked during updates to the structure, which occurs during the data ingestion phase.

D. *Paper Contributions and Translational Impact*

This study describes our methodology to support spatiotemporal sliding window queries at scale over voluminous, spatiotemporal gridded datasets. We place no constraints on the spatial resolution of these datasets nor the rate at which data are generated at each time step. Further, our query evaluations are amenable to supporting spatiotemporal intersections based on shapefiles from other data collections. Our contributions include:

1) A novel algorithm to support spatiotemporal sliding window queries over voluminous gridded datasets. Our methodology places no restrictions on the spatiotemporal resolutions associated with the gridded datasets nor the spatiotemporal scope associated with sliding window queries. Together, our data structure and algorithm for query evaluations reduce latencies and preserve throughputs. **[RQ-2]**
2) Our methodology includes a multi-pronged approach to scaling that encompasses memory residency, building sketches for dyadic intervals, and targeted replication schemes that account for query evaluation loads over particular dyadic intervals. **[RQ-2, RQ-3]**
3) Elimination of duplicate processing when aggregating observations into spatial extents – defined using N-sided polygons with each vertex encoded as a (latitude, longitude) pairs. Crucially, we support aggregation and disaggregation along administrative boundaries. **[RQ-1]**

Translational Impact: Our proposed methodology scales and is applicable to other gridded datasets. Gridded datasets continue to be made available in several domains. In particular, our algorithms and data structures could be used in settings where distances are based on traditional Cartesian coordinates and time is, unlike the spherical coordinates in the spatial referencing system we consider here, a lot more fine-grained – for example, gridded simulations of phenomena such as fluid dynamics.

More broadly, because it allows interactions between data represented in other formats such as shape files, point observations, etc. our methodology allows analyses to be performed over diverse data formats. Several domains such as ecology, environmental modeling, disaster and infrastructure planning rely on atmospheric and meteorological data (historical, current, and future) that are encoded in gridded formats to inform decision making. By allowing (dis)aggregation if matching spatial extents along administrative boundaries down to the census-tract level our methodology can interoperate with schemes that rely on such agglomeration schemes.

E. *Paper Organization*

The remainder of this paper is organized as follows. In Section 2, we provide an overview of the related work in the area. Our methodology is outlined in Section 3. Performance benchmarks alongside a description of datasets and discussion of results is outlined in Section 4. Finally, our conclusions and future work are described in section 5.

II. Related Work

Probabilistic algorithms and data structures enable space efficient data processing and generally require only a single pass over the data as it arrives. Count-min (CM) provides event frequencies using sublinear memory space, where an event could be a particular feature value or observation [1] [2]. CM is closely related to Bloom filters, which employ hash functions over a fixed-size bit array to determine set membership. With Bloom filters, false positives are possible but false negatives are not [3]. Several streaming algorithms have been developed to determine the number of distinct (unique) elements in a multiset, such as HyperLogLog++ [4], HyperLogLog [5], LogLog [6], and Linear Counting [7]. These probabilistic structures answer custom queries such as set membership, cardinality, etc. but are not suited for general processing. Wavelets create high-fidelity approximations of underlying observations but require problem-specific tuning [8] [9]. This effort supports expressive queries aligned with researcher needs.

Data storage systems leverage indexing as a key construct to assist in effective data retrievals and alleviate overheads involved in join operations. Each index consumes disk space and depending on the column being indexed the index may end up being very large. R-Trees [10], geohashes [11], and

quad-tiles [12] have been used to partition and organize spatial datasets. These structures are typically used to inform data dispersion schemes and not to evaluate complex query predicates over sliding temporal windows.

Scientific Data Management SciDB [13] [14] is a science-oriented database that supports multidimensional arrays in a shared-nothing architecture. The Data Capacitor [15] project relies on using the Lustre file system [16] and WAN to provide access to voluminous datasets. These systems are primarily geared towards data access and not towards real-time discovery and transformations or analytics over high-dimensional datasets. Time-series databases such as Prometheus [17], InfluxDB [18], KairosDB [19], OpenTSDB [20], and others [21] support storing time-series data. These efforts include mechanisms for horizontal scalability, time-based indexing of content, and support for custom query languages. Other commonly used storage systems typically used by researchers include distributed file systems with support for traditional directory structures and file layouts, such as HDFS [22] or NFS [23], and distributed hash tables (DHTs) that are organized as an overlay network and implement key-value storage including Pastry [24], PAST [24], and problem-specific frameworks such as Galileo [25]. Document-oriented databases such as MongoDB [26], CouchDB [27], and OrientDB [28] provide storage for objects including JSON documents or XML. The Griddle framework, and the encompassing GeoSieve data structure, that we describe here are focused on highly space-efficient (memory-resident) data structures with at least one instance per dyadic interval alongside support for temporal anchoring of query predicates to allow chaining of queries over GeoSieve instances to support arbitrarily sized temporal windows.

III. Methodology

Our methodology includes a set of phases that work in concert with each other to support query evaluations at scale. In particular, this includes: (1) Support for ingestion and data preprocessing, (2) an indexing data structure, or sketch, GeoSieve that is constructed from the voluminous data for different dyadic intervals, (3) an algorithm for query decomposition and distributed evaluation of queries, and (4) support for targeted replication of GeoSieve data structure instances.

A. *Ingestion and Data Preprocessing [RQ-1]*

Our framework can interface(?) with diverse distributed storage management frameworks such as HBase, Druid, MongoDB, etc. We chose to use the Apache Druid distributed database on top of HDFS to stage data for our experiments, as it appeared well-suited for handling highly voluminous data [29].

Most gridded datasets are provided in netCDF format, which is a compressed binary format that cannot be directly ingested into most databases, including Druid. We designed a custom netCDF to CSV converter to handle this conversion, then ingested the resulting CSVs into Druid.

To perform a mapping from the raw (latitude, longitude) points of each observation in the dataset to discretized spatial regions, we pre–compute expensive polygon lookups and store the results using an efficient Bloom Filter based in–memory scheme. Bloom Filters are probablisitc set membership data structures; instances may have occassional false positives, but are guaranteed to never have a false negative. In the pre–computation phase, we extract each of the unique (latitude, longitude) points from a given dataset then perform a polygonal query against a collection of shapefiles to determine which administrative boundary it lies within. After mapping each point to a region, Bloom Filters are created for each region containing a list of (latitude, longitude) points which are known to be in the region. These filters are then stored in an in-memory key–value database, such that truncated points are mapped to sets of regions that might contain those points and regions are mapped to Bloom Filters containing all of the full-resolution (latitude, longitude) points which are in that region.

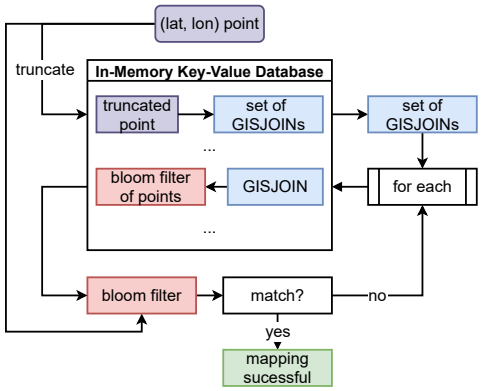

Fig. 1: The data structure and process used to speed up administrative boundary lookups during ingestion. An in-memory key-value database stores bloom filters keyed by administrative boundaries (GISJOINs) as well as sets of administrative boundaries keyed by a truncated version of the point they are in. A point is truncated and looked up to see what potential boundaries it lies in, then each of those boundaries is tested against the bloom filter. The database itself is populated by polygon intersection queries beforehand.

Method	Average lookups / sec	Standard deviation
Redis bloom filter lookups	39983.140	7388.363
MongoDB polygon lookups	283.364	107.535

TABLE I: The throughput observed from performing 1 minute of single-threaded lookup operations with our Bloom Filter strategy on a Redis cluster of 15 machines vs the same lookup operations using simple polygon lookups on a MongoDB cluster of 40 machines.

The process of mapping (latitude, longitude) points to administrative regions at ingestion time is illustrated in Figure 1. First, we truncate the point's coordinates and look up this truncated point in our in–memory collections. This will return a set of administrative regions (GISJOINs) in which the point might exist. Then, for each region R, we look up R in our collections, giving us a Bloom Filter containing the set of full-precision (latitude, longitude) points that exist in R. We search

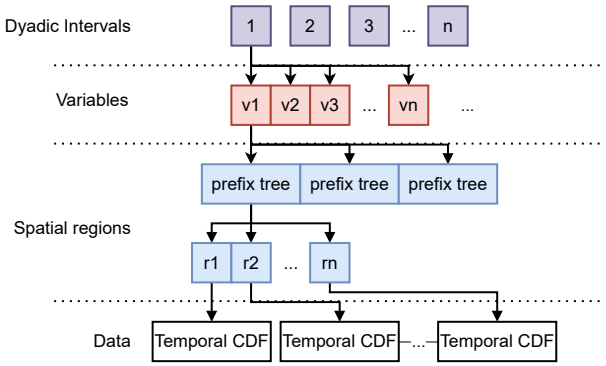

Fig. 2: A high–level view of the GeoSieve data structure. Each level of the tree is a set of index nodes . The middle level is a map of variables (v1, v2, ... vn) to prefix trees whose leaves are temporal CDFs of the parent variable. The prefix trees resolve the spatial component of the data (the administrative region) and have one leaf per region (r1, r2, ... rn) in the dataset.

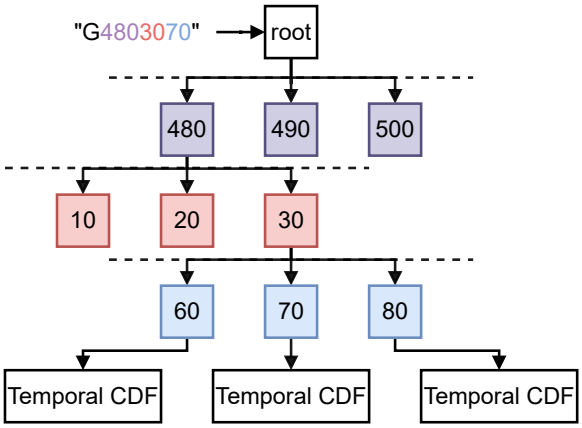

Fig. 3: The structure of the spatial resolving prefix tree. Our representation of administrative regions is a 7-digit number, which is split into 3 groups of digits, one for each level of the tree. The next node to follow on each individual level is resolved via a hash table.

the bloom filter for a match with our original point. If there is a match, we know the point is in R; otherwise, move on to the next region. If every point in the dataset is in some R, then this mapping will never fail, since Bloom Filters will never report a false negative as to whether an item exists or not.

We used a Redis cluster of 15 machines as our in–memory store for this mapping process. To evaluate its performance, we compared the amount of mappings per second achievable by our system vs. those achievable by performing simple polygon lookups in a traditional database. As can be seen from the results depicted in Table I, our lookup process has significantly higher throughput in contrast with the standard within-polygon query (`$geoIntersects`) in MongoDB.

B. GeoSieve Data Structure [RQ-3]

Conceptually, the GeoSieve data structure is a collection of maps that associate values of a variable to sets of timestamps. For each variable in the dataset, we choose a set of valid values for that variable, and for each of these values, we build a set of timestamps containing each time at which the value was contained between the minimum and maximum values of a window which starts at that time. Each of these maps is specific for one variable, spatial region, and window size. These maps are placed at the leaves of the GeoSieve tree, and the index nodes help locate them for a given window size, variable, and spatial region. There are 3 levels of indexing within the GeoSieve tree: the top level resolves the window size, the second level resolves the variable, and the third level resolves the spatial region. Figure 2 illustrates this structure.

To store the timestamp sets, timestamps are mapped to integers in a dataset-specific process. We choose some temporal granularity that the dataset can support, such as days or hours, and map the first of these units in the dataset to 0, the next to 1, and so on. The set of timestamps for each variable value then becomes a set of integers. We can represent this set of integers as a bitset, where the presence of a 1 in position n represents the timestamp for time unit n being included. These bitsets are often highly sparse and amenable to compression; we compress them with run–length encoding to significantly reduce their memory footprint.

For each variable, we must discretize the space of values to act as keys for the timestamp sets. Our implementation chooses 100 values spaced evenly between the minimum and maximum values of the variable observed in the dataset.

One such timestamp set is created for every variable and spatial region in the dataset. We use a hierarchical representation of administrative boundaries to store these spatial regions. To support efficient but generic spatial lookups, we store them as leaves of a prefix tree, as shown in Figure 3.

Query Structure and Process Queries to the GeoSieve structure are represented by one or more predicates and a sliding temporal window size. Each predicate is a tuple of a variable name, a value of interest (called the pivot), and a '↑' or '↓' identifier. If '↑' is used, the predicate will be true for times and locations where windows of the requested size saw all observations above the given pivot; if '↓' is used, all observations must be below the pivot. Optionally, queries are also permitted to limit their spatial or temporal scope by accepting a list of regions and/or timestamps by which to filter results. Algorithm 1 details the process of evaluating a single–predicate query against a GeoSieve instance.

The result of a query is a set of pairs containing a region and a single timestamp where, given a sliding window of the requested size starting at this time, every observation in the window satisfied the predicates.

A query may have multiple predicates. Each of these predicates can be evaluated as entirely independent sub-queries over the same GeoSieve instance, allowing them to execute concurrently. The query must specify whether it is interested in seeing the intersection or union of the results of the subqueries.

Thread Safety During queries, the GeoSieve data structure is thread safe. Since traversing the tree does not mutate any of its state, nor require any external shared, mutable state, queries

Algorithm 1 Single-predicate query evaluation algorithm

T_0 : Root of GeoSieve tree
p_s : Dyadic interval size
p_v : Variable being queried over
p_r : Set of regions to evaluate over
p_t : Predicate type ($\downarrow$ or $\uparrow$)
p_p : Predicate pivot
function EXECUTEQUERY($T_0, p_s, p_v, p_r, p_c, p_t, p_p$)
  $results \leftarrow \emptyset$
  $T_1 \leftarrow T_0\,[p_s]$ ▷ resolve interval size
  $T_2 \leftarrow T_1\,[p_v]$ ▷ resolve variable
  for r in p_r **do**
    $leaf \leftarrow T_2[r]$ ▷ resolve region
    if p_t is $\downarrow$ **then**
      $V \leftarrow$ the set of discretized values lower than p_p
    else
      $V \leftarrow$ the set of discretized values higher than p_p
    end if
    $R \leftarrow \emptyset$
    for v in V **do**
      $T \leftarrow leaf.get\_timestamp\_set\_for(v)$
      $R \leftarrow R \cup T$
    end for
    $results \leftarrow results \cup R$
  end for
  return $results$
end function

require only the lightweight acquisition of a shared lock. This allows for highly concurrent evaluation of queries over single GeoSieve instances.

New data may be added at any time to an existing GeoSieve instance. However, during such updating, the tree's state is mutated and synchronization is therefore required. We use a Reader–Writer lock to ensure thread–safety in the face of possible updates: querying threads acquire a read lock and updating threads acquire a write lock, thus guaranteeing that queries only block if an update is being performed. If updates are relatively infrequent, query evaluations will remain highly concurrent.

C. *Decomposition and Distributed Evaluation of Queries*

Queries may specify temporal sliding windows of any size, but each GeoSieve instance summarizes the dataset for one window size only and can therefore only handle queries of their exact size. To avoid instantiating a unique GeoSieve structure for every conceivable sliding window size, we instead instantiate at dyadic intervals and decompose queries to access multiple intervals.

A GeoSieve instance is constructed for each power of 2 up to a certain size. When a query is made, the provided window size is broken into its dyadic intervals. For instance, a window size of 13 is made of the dyadic intervals 8, 4, and 1. Then, for each interval i in descending order, we send the query to GeoSieve$\langle 2^i \rangle$, gather the spatiotemporal scopes returned by the query, and then use those spatiotemporal scopes as anchor points for the next query. In our example of 13, we would first raise a query against GeoSieve$\langle 2^3 \rangle$, then, if any results were returned, use them as the anchor for a query of the same predicates to GeoSieve$\langle 2^2 \rangle$, and finally if any results were returned, use them as the anchor for a query to GeoSieve$\langle 2^0 \rangle$. This is illustrated by part A of Figure 4. This method allows us to perform a query for any sliding window size smaller than 2^n, where n is the number of GeoSieve instances we construct. **Therefore, to support queries of up to size w, we need only construct $\lceil \log_2(w) \rceil$ GeoSieve instances.**

Algorithm 2 Dyadic interval distribution algorithm

q_s : Size of sliding window query
q_p : Predicates of sliding window query
function DISTRIBUTEQUERY(q_s, q_p)
  $results \leftarrow \emptyset$
  for b in the bits of q_s in descending order **do**
    if b is 1 **then**
      if $results$ is not $\emptyset$ **then**
        $scopes \leftarrow spatial\_scopes\_in(results)$
        $constrain\_next\_query\_to(scopes)$
      end if
      $i \leftarrow$ the index of the bit b
      $node \leftarrow get\_cluster\_node\_for\_interval(2^i)$
      $results \leftarrow results \cap node.query(q_p,\ regions)$
    end if
  end for
  return $results$
end function

In this way, we can also support distributed evaluation of queries by splitting the GeoSieve instances over multiple machines. In the previous example, it may be the case that GeoSieve$\langle 2^3 \rangle$, GeoSieve$\langle 2^2 \rangle$, and GeoSieve$\langle 2^0 \rangle$ are each on separate machines. Any such machine may accept a query and act as a coordinator by deconstructing the query, sending the pieces to each appropriate machine, and collecting the results, as illustrated in Figure 5 and detailed in Algorithm 2.

D. *Targeted Replication of GeoSieve Instances [RQ-3]*

We include support for targeted replication schemes allowing GeoSieve instances to be scaled commensurate with the observed access patterns. Upscaling and downscaling of replica instances is informed by our ingress matrix which tracks the number of accesses to GeoSieve instances as queries are evaluated. Time is partitioned into slices and represented as rows within our matrix. We allow users to configure the number and duration of time slices. The default configuration of the ingress matrix captures accesses for the last 60 time-slices where each time-slice is roughly 1 minute. The matrix is complemented with a row pointer representing the start of a new slice and the row that represents the current time slice. The row pointer is incremented at the end of every time slice to indicate the current time slice i.e. $\texttt{row\_pointer} = (\texttt{row\_pointer} + 1) \bmod (N-1)$. This allows us to represent

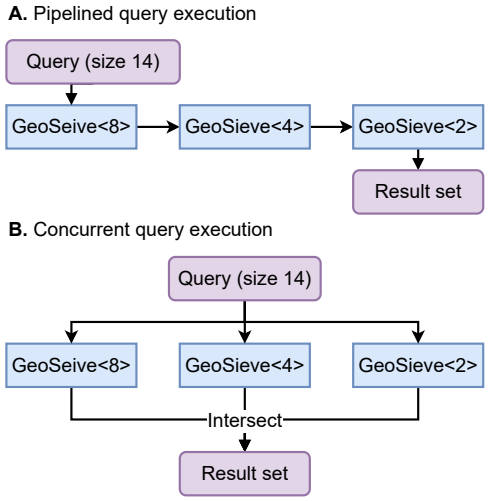

Fig. 4: The two methods by which a query requiring multiple dyadic intervals may be presented to instances of the GeoSieve data structure. This example query has a sliding window size of 14, meaning it requires three intervals to be traversed (2, 4, and 8). In pipelined query execution, the query is presented to each instance in sequence, from largest interval to smallest, with the results from the previous instance anchoring the queries on the next. In concurrent query execution, we run the query on each instance concurrently, intersecting each result set together to get the total result set.

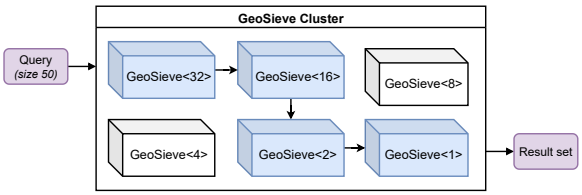

Fig. 5: An example of evaluating a query across a GeoSieve cluster. Each machine contains one or more instances of the GeoSieve structure over different dyadic intervals. Any arbitrary machine in the cluster may accept a query and act as a coordinator for it, routing it to the correct sequence of machines. The coordinator will break down the query by its dyadic intervals and send it to each of the necessary machines in a pipelined fashion, until all GeoSieve instances required by the query have been processed; then the results return to the coordinator and are reported.

the matrix as a circular buffer, where the row representing oldest time slice (60 time slices prior) is overwritten by data representing access counts that are occurring during the current time slice.

The matrix contains M columns with each column representing dyadic intervals: $2^0, 2^1, 2^2, \ldots, 2^{M-1}$. Each column tracks the number of accesses to the GeoSieve for a dyadic interval. A cell (i, j) within this matrix represents the number of accesses for a particular time-slice i for dyadic interval 2^i. Row_i can be used to estimate the total number of accesses to all dyadic intervals for a time-slice. The sum of the entries within a column can be used to estimate accesses to a particular dyadic instance for the time-frame (default 60 minutes) representing the most recent N time-slices (each 1 minute).

We use the matrix to estimate the total number of accesses per-dyadic interval for the last N (default, 60) time slices. We use these estimates to inform proportional allocation of replica instances of the GeoSieve data structure for particular dyadic intervals, as detailed in Algorithm 3. The re-replication maneuvers are performed at a user-specified duration in increments of 15 time-slices: a larger time scale is chosen to avoid oscillatory behavior in the system.

Algorithm 3 Targeted replication algorithm

Require: $|N|$ is divisible by $(n_x - n_s)$
N : List of nodes in cluster
n_x : Maximum desired dyadic interval size
n_s : Minimum desired dyadic interval size
M : Dyadic interval usage matrix
function GETREPLICATIONSCHEME(N, n_x, n_s, M)
$n \leftarrow n_x - n_s$
$R \leftarrow$ array of n empty lists
for $i = n_s$ to $n_x - 1$ **do**
Add $N[i]$ to $R[i]$
end for
$U \leftarrow$ array of normalized sums of M's columns
for $i = 0$ to $n - 1$ **do**
Add next $\lfloor U[i] \times (|N| - n) \rfloor$ items of N to $R[i]$
end for
if N is not empty **then**
Add remaining elements of N to $R[n_s]$
end if
return R
end function

IV. Performance Evaluation and Discussion

In this section, we evaluate the latency, throughput, and memory pressure of querying the GeoSieve structure, both in single—machine and distributed environments.

A. *Experimental Setup and Datasets*

Our empirical benchmarks were conducted on HPE ProLiant DL60 Gen9 machines with 6-core 2.4 GHz CPUs and 64 GB of memory, running AlmaLinux 8.6 with kernel version 4.18.0. All software was implemented in Java using OpenJDK 11.

Two multidecade, widely–used gridded climate datasets were used to profile our methodology: gridMET and MACA.

a) gridMET is dataset of historical meteorological observations collected at regular points over the contiguous US. It includes daily readings of 15 variables such as air temperature, specific humidity, evapotranspiration, and wind speed, for roughly 500,000 geolocations across the contiguous US over a period of 43 years (1979-yesterday) at a daily temporal resolution.
For performance evaluations, we ingested gridMET containing all data from years 1979-2020. Uncompressed, the dataset is approximately 1.1 TB large.

b) The Multivariate Adaptive Constructed Analogs (MACA) datasets are sets of gridded data that represent possible future projections of Earth's climate. The datasets include 10 variables such as air temperature, specific humidity, wind speed/direction, and surface radiation for roughly 500,000 points across the US projected up to 80 years in the future (2100) at a daily temporal resolution. The data model two future scenarios, or Representative Concentration Pathways; one where an additional 4.5 W/m2 is trapped in the earth-atmosphere system by 2100 compared to preindustrial predicates (known as RCP4.5), and a more pessimistic scenario where an additional 8.5 W/m2 is trapped (known as RCP8.5). RCP4.5 roughly represents a future with moderate climate mitigation action and decreased emissions, while RCP8.5 represents a future with business as usual and high emissions. For our benchmarks, we ingested a subset of the GFDL-ESM2M model at the 8.5 scenario, ranging 30 years (2021-2050). Uncompressed, the dataset is approximately 530 GB large.

B. Systems Benchmarks

Our systems benchmarks profile several key aspects of our methodology. In particular, these include: (1) How well does the system cope with constructing and updating the GeoSieve data structure? A key measure here is the rate (or throughput) at which the data structure is able to ingest new, multivariate observations. (2) The GeoSieve data structure serves as surrogate (or sketch) for the on-disk data. The structure includes lightweight concurrency control and locking mechanisms, and support for traversals as queries are evaluated. We profile the memory footprint of the data structure over billions of multivariate observations encompassing the continental United States over multidecadal datasets. (3) How well does the methodology support queries that include multiple predicates over arbitrarily sized temporal windows over the entire spatial extent encapsulated by the dataset? We are interested in both the latency and throughput of the query evaluations.

Throughput of Building & Updating GeoSieve We evaluated performance, in terms of the number of pre–computed sliding window observations that can be ingested per second, of constructing a single GeoSieve instance for one dyadic interval. The construction process ran over nine simultaneous threads, and the throughputs observed for each thread were aggregated into Table II. We captured only the time it took for the data structure itself to ingest and process each observation, excluding the network I/O associated with transmitting observations to the data structure. These results reflect the performance of updating the data structure with new observations, as the process of constructing and updating are identical.

As depicted in Table II, GeoSieve construction and updating is efficient: gridMET observations have 15 variables, while MACA observations have 10. With nine threads operating concurrently, it is possible to achieve construction / update rates of over 40,000 observations per second. **[RQ-3]**

Memory Pressure We evaluated the resource overhead of loading GeoSieve instances built from our test datasets into memory. Table III shows these results.

Dataset	Average observations / sec	Standard deviation
gridMET	4558.669	821.044
MACA	8385.071	1272.269

TABLE II: The average single-threaded throughput observed from building a single instance of GeoSieve for both of our test datasets, in terms of the number of observations ingested per second.

Dataset	Dyadic interval size	Memory usage
gridMET	2^1	8.723 GB
gridMET	2^2	5.363 GB
gridMET	2^3	2.735 GB
gridMET	2^4	1.518 GB
gridMET	2^5	1.082 GB
MACA	2^1	3.435 GB
MACA	2^2	2.200 GB
MACA	2^3	1.314 GB
MACA	2^4	0.883 GB
MACA	2^5	0.694 GB

TABLE III: The amount of resources consumed by loading differently-sized GeoSieve instances into memory for both of our test datasets.

We observe a memory footprint squarely under 10 GB for both of our datasets for each of the tested dyadic interval sizes ($2^1, 2^2, 2^3, 2^4, 2^5$). The smaller of our two test datasets, MACA, is also accordingly significantly smaller in memory. These results illustrate that it is entirely possible to load one or multiple GeoSieve instances into the memory on a single machine, demonstrating that we are able to avoid costly disk and/or network I/O when traversing the structure, especially when dyadic intervals are split among multiple machines. [**RQ-2**, **RQ-3**]

Query Throughput and Latency We evaluated the throughput and latency of the data structure by running queries on a cluster of GeoSieve instances containing dyadic intervals $2^1, 2^2, 2^3, 2^4$, and 2^5 for both of our test datasets. To build a suite of test queries, we enumerated over the Cartesian product of 3 factors: the number of dyadic intervals (1–5), the number of predicates in the query (1–5), and the method of composing the predicates in the query (either AND or OR). Predicates were constructed by examining extreme cases of the variables of interest; for example, predicates involving precipitation looked for regions where rainfall was less than 0.01 mm. The result sets of our queries contained around 4000-5000 spatiotemporal points on average. The throughput measurements for both of our test datasets are shown in Figures 6 (gridMET) and 7 (MACA). Maximum throughput for each cluster size was measured by steadily raising the query submission rate until the amount of queued queries exceeded the submission rate per second over a 1 minute

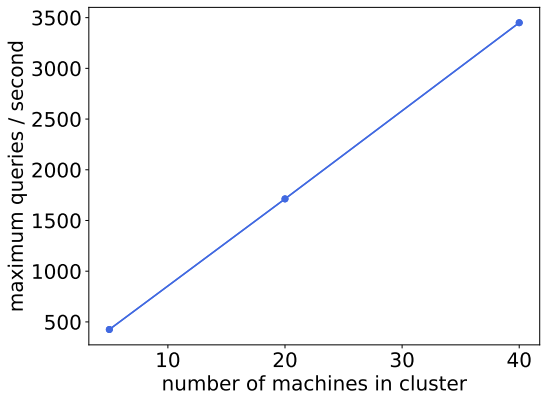

Fig. 6: The number of machines in our GeoSieve cluster vs. the maximum throughput we observed the cluster could support. Cluster sizes of 5 machines, 20 machines, and 40 machines were tested. Maximum queries per second for each cluster were measured by steadily raising the query submission rate until the amount of queued queries exceeded the submission rate per second at the end of a 1 minute period of continuous queries.

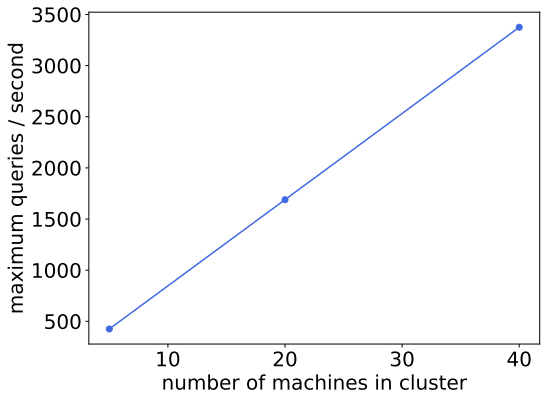

Fig. 7: Figure 6, but considering MACA data instead of gridMET data. Note the significant similarity in scaling performance.

period. The latency measurements for both of our test datasets are shown in Tables IV and V.

The throughput of our system scales linearly with the number of machines in the cluster. There also appears to be no significant difference between our two test datasets in terms of query throughput, despite the difference in size between them. These results show that highly concurrent query evaluations at arbitrary scales are possible with the GeoSieve system **[RQ-3]**.

Tables IV and V show that querying the GeoSieve structure is extremely fast. Sliding window queries over the entirety of the summarized dataset take less than 15 ms to complete, even with up to five query predicates. In addition, querying multiple dyadic intervals does not appear to result in a performance degradation, suggesting the overhead of switching between multiple GeoSieve instances to complete a query is not large enough to overcome other factors, such as the number of predicates or number of results produced by the query. This

Dataset	Predicates in query	Average query time (ms)
gridMET	1	2.558
gridMET	2	4.846
gridMET	3	7.292
gridMET	4	9.456
gridMET	5	11.251
MACA	1	2.278
MACA	2	4.136
MACA	3	6.015
MACA	4	8.058
MACA	5	10.249

TABLE IV: The average latency observed from querying our test datasets in a single–machine environment, aggregating the results by the number of predicates in the query. On average, even queries with up to 5 predicates have under 15 ms of latency. MACA is slightly faster than gridMET for all predicate counts.

Dataset	Dyadic intervals hit	Average query time (ms)
gridMET	1	7.155
gridMET	2	5.342
gridMET	3	3.796
gridMET	4	4.764
gridMET	5	5.326
MACA	1	5.586
MACA	2	4.570
MACA	3	3.527
MACA	4	4.483
MACA	5	4.917

TABLE V: The average latency observed from querying our test datasets in a single–machine environment, aggregating the results by the number of predicates in the query. On average, even queries with up to 5 predicates have under 15 ms of latency. MACA is slightly faster than gridMET for all predicate counts.

shows query evaluations over the GeoSieve structure are timely enough to support real–time analysis even in a non-clustered environment. **[RQ-2, RQ-3]**

V. Conclusions & Future Work

Here, we described our methodology to support effective management of voluminous gridded datasets alongside effective evaluation of queries of arbitrary temporal lengths.

RQ-1: We map lat-long grid points to spatial extents based on hierarchical prefixes that are amenable to deterministic and decentralized prefix-based aggregation. A shorter-prefix corresponds to a large spatial extent encompassing multiple, contiguous spatial extents with shared prefixes, but a longer length. For example, this prefix-based scheme allows all data from multiple encompassing census tracks within a county to be colocated on the same machine if prefix lengths associated with counties is chosen as the unit of dispersion. The use of Bloom Filters allows us to quickly assess set memberships without incurring duplicate evaluation costs associated with

testing for inclusion of grid points within N-sided shapes. Our scheme performs over 100x faster than such polygon lookups in a single-threaded environment.

RQ-2: The GeoSieve data structure design is aligned with the nature of sliding–window query evaluations. The data structure is designed as a tree with each level providing feature-specific evaluation of query predicates. Rather than support arbitrary window sizes, we create GeoSieve instances that evaluate over dyadic intervals with at least one instance of GeoSieve per dyadic interval. GeoSieve includes support for anchoring queries with chronological starting points. Queries are transformed so that predicate evaluations represent traversals of the tree structure. A query is decomposed into dyadic intervals and our runtime accounts for daisy-chaining queries over appropriate dyadic instances and anchoring query evaluations with the correct spatiotemporal bookends retrieved upstream from the chain. Crucially, the tree traversals and dyadic intervals allow us to identify early-stopping criteria for queries whose predicates evaluate to false. Finally, GeoSieve instances are highly memory efficient and incorporate support for run length encoding to further reduce memory footprints associated with the storing temporal information in the leaves. Datasets up to 1 TB in uncompressed size can be represented as a GeoSieve sketch in under 10 GB of memory, and under 2 GB of memory for larger dyadic interval sizes.

RQ-3: The GeoSieve data structure includes a novel mix of data structure design, targeted replications, temporal anchoring, query transformations, and locking and concurrency schemes. There is one GeoSieve instance per dyadic interval and each instance can be replicated independently of the other. Each instance returns the set of spatial extents (and the temporal bounds) over which the query predicates evaluated to true – this allows us to chain queries across different dyadic intervals. This is backed by a scheme that allows to compute intersections of spatiotemporal scopes returned by query evaluations. Our methodology allows multiple, concurrent query evaluations to be performed. Updates to the data structure involve exclusive locking and these are performed periodically or in batches. This reduces lock contention and allows the query evaluations to scale. The throughput of query evaluations scales linearly with the number of GeoSieve instances in a machine cluster.

Our future work will explore support for interactive visualization of gridded datasets. This will necessitate support for distributed caching schemes that include support for speculative prefetching, and evictions, based on visualization patterns and exploration trajectories. A key goal is to ensure that disk I/O is not in the critical path of data visualizations.

ACKNOWLEDGMENT

This research was supported by the National Science Foundation [OAC-1931363, ACI-1553685] and the National Institute of Food & Agriculture [COL0-FACT-2019].

REFERENCES

[1] G. Cormode, "Count-min sketch." 2009.

[2] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, 2013, pp. 683–692.

[5] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[6] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *European Symposium on Algorithms*. Springer, 2003, pp. 605–617.

[7] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, 1990.

[8] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine *et al.*, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends® in Databases*, vol. 4, no. 1–3, pp. 1–294, 2011.

[9] S. Yousefi, I. Weinreich, and D. Reinarz, "Wavelet-based prediction of oil prices," *Chaos, Solitons & Fractals*, vol. 25, no. 2, 2005.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, 1990, pp. 322–331.

[11] G. Niemeyer, "Geohash: wikipedia.org/wiki/geohash," 2008.

[12] OSM, "Quadtiles: Geodata storage and indexing," 2022. [Online]. Available: https://wiki.openstreetmap.org/wiki/QuadTiles

[13] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla *et al.*, "A demonstration of scidb: a science-oriented dbms," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1534–1537, 2009.

[14] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *International Conference on Scientific and Statistical Database Management*. Springer, 2011, pp. 1–16.

[15] S. C. Simms, G. G. Pike, and D. Balog, "Wide area filesystem performance using lustre on the teragrid," Tech. Rep., 2007.

[16] W. Yu, R. Noronha, S. Liang, and D. K. Panda, "Benefits of high speed interconnects to cluster file systems: a case study with lustre," in *IEEE International Parallel & Distributed Processing Symposium*, 2006.

[17] "Prometheus - monitoring system and time series database," 2019. [Online]. Available: https://prometheus.io

[18] "Influxdb: Time series database. real-time visibility into stacks, sensors and systems." 2018. [Online]. Available: https://influxdata.com

[19] "Kairosdb: Fast time series database on cassandra." 2018. [Online]. Available: https://kairosdb.github.io

[20] "Opentsdb - a distributed, scalable monitoring system," 2019. [Online]. Available: http://opentsdb.net

[21] R. Stephens, "The state of the time series database market," 2018. [Online]. Available: https://redmonk.com/rstephens/2018/04/03/the-state-of-the-time-series-database-market/

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[23] B. Nowicki, "Transport issues in the network file system," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 2, 1989.

[24] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[25] M. Malensek, S. Pallickara, and S. Pallickara, "Analytic queries over geospatial time-series data using distributed hash tables," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 6, 2016.

[26] K. Banker, D. Garrett, P. Bakkum, and S. Verch, *MongoDB in action: covers MongoDB version 3.0*. Simon and Schuster, 2016.

[27] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide: time to relax*. " O'Reilly Media, Inc.", 2010.

[28] C. Tesoriero, *Getting started with OrientDB*. Packt Publishing Birmingham, England, 2013.

[29] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *ACM SIGMOD international conference on Management of data*, 2014, pp. 157–168.