ChronosDB: Distributed, File Based, Geospatial Array DBMS

Ramon Antonio Rodriges Zalipynis
National Research University Higher School of Economics, Moscow, Russia
arodriges@hse.ru

ABSTRACT

An array DBMS streamlines large N-d array management. A large portion of such arrays originates from the geospatial domain. The arrays often natively come as raster files while standalone command line tools are one of the most popular ways for processing these files. Decades of development and feedback resulted in numerous feature-rich, elaborate, free and quality-assured tools optimized mostly for a single machine. ChronosDB partially delegates in situ data processing to such tools and offers a formal N-d array data model to abstract from the files and the tools. ChronosDB readily provides a rich collection of array operations at scale and outperforms SciDB by up to 75× on average.

PVLDB Reference Format:
R.A. Rodriges Zalipynis. ChronosDB: Distributed, File Based, Geospatial Array DBMS. *PVLDB*, 11(10): 1247-1261, 2018.
DOI: https://doi.org/10.14778/3231751.3231754

1. INTRODUCTION

A georeferenced raster or more generally an N-d array is the primary data type in a broad range of fields including climatology, geology, and remote sensing which are experiencing tremendous growth of data volumes. For example, DigitalGlobe is a commercial satellite imagery provider that collects 70 TB of imagery on an average day [35].

Traditionally rasters are stored in files, not in databases. The European Centre for Medium-Range Weather Forecasts (ECMWF) has alone accumulated 137.5×10^6 files sized 52.7 PB in total [26]. This file-centric model resulted in a broad set of sophisticated raster file formats. For example, GeoTIFF is an effort by over 160 different remote sensing, GIS (Geographic Information System), cartographic, and surveying related companies and organizations [23]. NetCDF has been under development since 1990 [56], is standardized by OGC [44], and supports multidimensional arrays, chunking, compression, and hierarchical name space [40].

Command line tools have long being developed to manage raster files. Many tools have large user communities that are very accustomed to them. NCO (NetCDF Operators) have been under development since about 1995 [43], GDAL (Geospatial Data Abstraction Library) provides tools for managing over 150 raster file formats, has $\approx 10^6$ lines of code and hundreds of contributors [14]. Many tools utilize multicore CPUs but mostly work on a single machine.

The in situ technique has gained increased attention due to the explosive growth of raster data volumes in diverse file formats [3, 85, 6, 8]. Unlike the in-db approach, the in situ approach operates on data in their original file formats in a standard file system and does not require importing into an internal DBMS format. Files may be preprocessed before querying but this is usually much faster than a full import.

Keeping data of an array DBMS in open, widely adopted, and standardized file formats has numerous advantages: easier data sharing, powerful storage capabilities, the absence of a time-consuming and error-prone import phase for many data types, direct accessibility of DBMS data to other systems, and easier migration to other DBMS to name a few [59].

Many raster processing algorithms require significant implementation efforts, but already existing stable and multifunctional tools are largely ignored in this research trend. Algorithms are re-implemented almost from scratch delaying the emergence of an array DBMS being competitive with existing operational pipelines, e.g. SciDB appeared in 2008 and still lacks even core operations like interpolation [65].

The idea of partially delegating in situ array operations to existing tools was first realized in ChronosServer [59, 58, 11, 60, 62, 61]. Its successor is ChronosDB extended with a significantly improved data model (section 2), new components, optimizations, array management, query execution capabilities (section 3), and array operations (section 4).

The ChronosDB formal data model uniformly represents diverse raster data types and formats, takes into account the distributed context, and is independent of the underlying raster file formats and tools at the same time (section 2).

ChronosDB distributed algorithms are based on the model, unified, formalized, and targeted at the in situ processing of arbitrary geospatial N-d arrays (section 4).

The algorithms are designed to always delegate significant portions of their work to the tools. This proves wide applicability of the tools and the delegation approach. The delegation happens by direct submission of files to a tool as command line arguments to process them on a single cluster node. ChronosDB re-partitions and streams input/output files between the nodes and tools to scale out the processing.

In summary, the major contributions of our work include: (1) we show that it is possible to build a well-abstracted,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.

Proceedings of the VLDB Endowment, Vol. 11, No. 10
Copyright 2018 VLDB Endowment 2150-8097/18/06... $ 10.00.
DOI: https://doi.org/10.14778/3231751.3231754

extensible, and efficient distributed geospatial N-d array DBMS by leveraging existing elaborate command line tools designed for a single machine;

(2) we give a holistic description of the complete DBMS leveraging the tools: ChronosDB has a relatively simple yet flexible coordination layer on top of powerful components. This enables exceptional performance, rich functionality and avoids re-implementing array algorithms from scratch. It is also easier to manage the development life cycle of such a DBMS as it is comprehensive by a small team of engineers;

(3) we thoroughly compare ChronosDB and SciDB on up to 32-node clusters in the Cloud on real-world data: Landsat satellite scenes and climate reanalysis. We took SciDB as it is the only freely available distributed array DBMS to date.

2. CHRONOSDB DATA MODEL

2.1 Motivation

The most widely used industry-standard data models for abstracting from raster file formats are Unidata CDM, GDAL Data Model and ISO 19123. Industrial models are mappable to each other [40] and have resulted from decades of considerable practical experience. However, they work with a single file, not with a set of files as a single array.

The most well-known research models and algebras for dense N-d, general-purpose arrays are AML [38], AQL [37], and RAM [81]. They are mappable to Array Algebra [4].

The creation of the ChronosDB data model was motivated by the following features that are not simultaneously present in the existing data models: (1) the treatment of arrays in multiple files arbitrarily distributed over cluster nodes as a single array, (2) formalized industrial experience to leverage it in the algorithms, (3) a rich set of data types (Gaussian, irregular grids, etc.), (4) subarray mapping to a raster file is almost 1:1 but still independent from a format. As can be seen from [40], the N-d array model (section 2.2) resembles CDM while the two-level set-oriented dataset model (section 2.3) provides additional necessary abstractions.

Many popular command line tools rely on CDM, GDAL or ISO. This makes the tools capable of handling data in diverse raster file formats and be readily compatible with the ChronosDB data model.

2.2 Multidimensional arrays

In this paper, an N-dimensional array (N-d array) is the mapping $A : D_1 \times D_2 \times \cdots \times D_N \mapsto \mathbb{T}$, where $N > 0$, $D_i = [0, l_i) \subset \mathbb{Z}$, $0 < l_i$ is a finite integer, and $\mathbb{T}$ is a numeric type[1]. l_i is said to be the *size* or *length* of ith dimension[2]. Let us denote the N-d array by $A\langle l_1, l_2, \ldots, l_N\rangle : \mathbb{T}$. By $l_1 \times l_2 \times \cdots \times l_N$ denote the *shape* of A, by $|A|$ denote the *size* of A such that $|A| = \prod_i l_i$. A *cell* or *element* value of A with integer indexes $(x_1, x_2, \ldots, x_N)$ is referred to as $A[x_1, x_2, \ldots, x_N]$, where $x_i \in D_i$. Each cell value of A is of type $\mathbb{T}$. A missing value is denoted by `NA`. An array may be initialized after its definition by enumerating its cell values. For example, the following defines and initializes a 2-d array of integers: $A\langle 2, 2\rangle : \text{int} = \{\{1, 2\}, \{\texttt{NA}, 4\}\}$. In this example, $A[0, 0] = 1$, $A[1, 0] = \texttt{NA}$, $|A| = 4$, and the shape of A is 2×2.

Indexes x_i are optionally mapped to specific values of ith dimension by *coordinate* arrays $A.d_i\langle l_i\rangle : \mathbb{T}_i$, where $\mathbb{T}_i$ is a totally ordered set, $d_i[j] < d_i[j + 1]$, and $d_i[j] \neq \texttt{NA}$ for $\forall j \in D_i$. In this case, A is defined as $A(d_1, d_2, \ldots, d_N) : \mathbb{T}$.

A *hyperslab* $A' \sqsubseteq A$ is an N-d subarray of A. The hyperslab A' is defined by the notation $A[b_1 : e_1, \ldots, b_N : e_N] = A'(d'_1, \ldots, d'_N)$, where $b_i, e_i \in \mathbb{Z}$, $0 \leqslant b_i \leqslant e_i < l_i$, $d'_i = d_i[b_i : e_i]$, $|d'_i| = e_i - b_i + 1$, and for all $y_i \in [0, e_i - b_i]$ the following holds: $A'[y_1, \ldots, y_N] = A[y_1 + b_1, \ldots, y_N + b_N]$, $d'_i[y_i] = d_i[y_i + b_i]$ (A and A' have a common coordinate subspace over which cell values of A and A' coincide). The dimensionality of A and A' is the same. We will omit "$: e_i$" if $b_i = e_i$ or "$b_i : e_i$" if $b_i = 0$ and $e_i = |d_i| - 1$.

Arrays X and Y overlap iff $\exists Q : Q \sqsubseteq X \wedge Q \sqsubseteq Y$. Array Q is called the *greatest common hyperslab* of X and Y and denoted by $gch(X, Y)$ iff $\nexists W : (W \sqsubseteq X) \wedge (W \sqsubseteq Y) \wedge (Q \sqsubseteq W) \wedge (Q \neq W)$. An array X covers an array Y iff $Y \sqsubseteq X$.

2.3 Datasets

Two dataset types exist: raw and regular. Raw datasets capture a broad range of possible raster data types, e.g. a set of scattered, overlapping satellite scenes, fig. 1. A raw dataset must be "cooked" into a regular one in order to perform array operations on it, section 3.3. Many real-world array data already satisfy regular dataset criteria and need not to be cooked, e.g. gridded geophysical data, section 5.

Both datasets are two-level and contain a user-level array and a set of system-level arrays (array and subarrays for short). Subarrays are distributed among cluster nodes and stored as ordinary raster files. An array is never materialized and stored explicitly: an operation with an array is mapped to a sequence of operations with respective subarrays.

Datasets are read-only: an array operation produces a new dataset. This has a strong practical motivation. The majority of raster data come from instrumental observation and numerical simulation. Original data are never changed once they are produced. Derivative data differ vastly: raster algorithms usually alter large portions of an array.

Formally, a *raw dataset* $\mathbb{D}^{raw} = (A, M, P^{raw})$ has a *user-level* array $A(d_1, d_2, \ldots, d_N) : \mathbb{T}$, metadata M, and a set of *system-level* arrays $P^{raw} = \{(A', B, E, M', wid)\}$, where $B\langle N\rangle : \text{int} = \{b_1, b_2, \ldots, b_N\}$, $E\langle N\rangle : \text{int} = \{e_1, e_2, \ldots, e_N\}$ such that $A' = A[b_1 : e_1, b_2 : e_2, \ldots, b_N : e_N]$, M' is metadata for A', wid is an identifier of a cluster node storing A', M and M' are sets of (name, value) pairs: M includes general dataset properties (name, description, etc.) and properties valid for $\forall p \in P^{raw}$ (type $\mathbb{T}$, storage format, etc.), e.g. $M = \{(type = \text{int16}), (format = \text{GeoTIFF})\}$, $M' = \{(date = \text{2015-Sep-08}, projection = \text{EPSG:32637})\}$.

A *regular dataset* $\mathbb{D} = (A, M, P, S, \rho, r^0)$ has a user-level array $A(d_1, d_2, \ldots, d_N) : \mathbb{T}$, metadata M, a set of system-level arrays $P = \{(A', B, E, M', wid, key)\}$, where A', B, E, M', wid mean the same as for P^{raw}, $key\langle N\rangle : \text{int} = \{k_1, k_2, \ldots, k_N\}$, $k_i \in \mathbb{Z}$, $A' \sqsubseteq A[h^b_1 : h^e_1, \ldots, h^b_N : h^e_N]$, where

$$h^b_i = max(r^0_i + k_i \times s_i - \rho_i, 0), \tag{1a}$$

$$h^e_i = min(r^0_i + (k_i + 1) \times s_i - 1 + \rho_i, |A.d_i| - 1), \tag{1b}$$

$S = (s_1, s_2, \ldots, s_N)$ is the largest possible shape for $\forall p \in P$, $\rho = (\rho_1, \rho_2, \ldots, \rho_N)$ is an overlap between subarrays, and $r^0 = (r^0_1, r^0_2, \ldots, r^0_N)$ is a reference index, $s_i, r^0_i \in \mathbb{Z}$, $s_i > 0$, $\rho_i \in [0, s_i \text{ div } 2) \subset \mathbb{Z}$, and $\nexists p, q \in P : p.key = q.key \wedge p = q$. Note that $\mathbb{D}^{raw}$ and $\mathbb{D}$ share A and M. Let us refer to subarray A' by key as $\mathbb{D}\langle\langle key\rangle\rangle$ or $\mathbb{D}\langle\langle k_1, k_2, \ldots, k_N\rangle\rangle$.

The above means that array A is divided by $(N-1)$-d hyperplanes on N-d subspaces, a subarray may not fully cover

[1]A C++ type according to ISO/IEC 14882 can be taken.

[2]Throughout this paper, $i \in [1, N] \subset \mathbb{Z}$

the subspace in which it resides, not all subspaces must contain a subarray, no two subarrays except their overlapping cells are inside the same subspace, subarray keys are unique. All cells of an empty subspace are equal to `NA`. In this way ChronosDB supports large sparse arrays, fig. 1.

Let us call $gch(A', A[h_1^b : h_1^e, \dots, h_N^b : h_N^e])$ the *body* of subarray A' provided that h_i^b and h_i^e are calculated given $\rho_i = 0$ for $\forall i$. The cells of a subarray that are not in its body are the *periphery* of this subarray. Let us refer to an element of a tuple $p = (A', B, \dots)$ in P^{raw} or in P as $p.A$ for A', $p.B$ for B, etc. Let $keys : \mathbb{D} \mapsto \{p.key : \forall p \in P\}$.

A regular dataset is shown in fig. 2c. Array $A(time, lat, lon)$ with shape $6\times2\times6$ is divided by 2-d planes into 9 subarrays (this is very common in practice), where $S = (2, 2, 2)$, overlap is not shown, $r^0 = (4, 0, 2)$. Subarrays with the same color reside on the same cluster node. The values of the coordinate arrays are shown next to each index. The key $\langle\!\langle -1, 0, 1\rangle\!\rangle$ refers to the subarray $A[2:3, 0:1, 4:5]$ for Jan 03-04, $lat = 20° \dots 30°$, $lon = 35° \dots 40°$.

3. CHRONOSDB ARCHITECTURE

3.1 Illustrative Dataset

To make this section clearer, we form a dataset as a running example: two mosaics of 4×8 Landsat 8 scenes, bands 4 (visible red, RED) and 5 (near-infrared, NIR) [33], paths 191–198, rows 24–27, 01–15 July 2015, GeoTIFF. Both in ChronosDB and SciDB, the mosaics are two 24937×38673 arrays: RED and NIR. An array size is ≈1.4 GB in SciDB and ≈ 1.6 GB in ChronosDB. Each array has ≈ 23% of empty cells (`NA`): the mosaic areas not covered by the scenes, fig. 1.

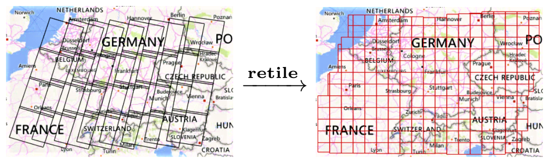


Figure 1: Footprints of scenes and subarrays, UTM 32N

3.2 ChronosDB Components

Gate and Workers run on a computer cluster of commodity hardware. A single Gate at a dedicated cluster node receives queries (scripts) and coordinates workers. Workers run at each node and are responsible for data processing.

Clients may connect to the Gate via WebSocket, establish a session, and communicate via a plain-text protocol: a Client ↦ Gate message carries a script while Gate ↦ Client messages carry ChronosDB output. A script is a sequence of one or more ChronosDB commands (section 3.4 gives an example). We implemented a JavaScript client in just 115 lines of code. Users can submit scripts via any third-party interface equipped with a similar client. Hence, users work with ChronosDB like with command line tools in a console.

ChronosDB Commands have the same syntax as command line tools. Command names coincide with the names of existing command line tools. Command options have the same meaning and names as tool options, but users specify input and output dataset names instead of file names. Users avoid learning a new language and work with ChronosDB as if with the console tools they are accustomed to with only minor changes to already familiar tool options. Most tool documentation is applicable to the respective ChronosDB command due to exactly the same meaning and behavior.

RAM Disk (RAMD) [31] is mounted on each worker. This allows the materialization of intermediate data to memory, often avoids hitting IOPS limits (I/O operations per second for SSD) in the Cloud, and yields large speedups (up to 3×).

Virtual file format (VFF) is a description of operations made to the source file [22, 42, 21]. A VFF often substantially reduces runtime space requirements and network I/O by avoiding intermediate materialized data (section 3.4).

Execution Pool can run up to c_p single-threaded tools in parallel. Each worker has such a pool with c_p slots, where c_p is the number of CPU cores.

3.3 Dataset Management

Consider a raw dataset $\mathbb{D}^{raw} = (A, M, P^{raw})$ and its regular derivative $\mathbb{D} = (A, M, P, S, \rho, r^0)$.

Dataset Namespace is hierarchical making it easier to navigate in a large number of datasets. It is similar to the ChronosServer namespace with over 800 registered datasets to date [11, 12]. For example, we refer to Band №4 of the illustrative dataset as `Landsat8.Level_1.SurfaceReflectance.Band4` (dots separate the names of dataset collections, the dataset name is the last). A Gate keeps the hierarchy and file naming rules in a human-readable XML file. For example, the Landsat scene file name `LC81910242015182LGN00_B4.TIF` contains its acquisition date: year (2015) and the day of this year (182) [34].

Subarray Metadata M' is in file names and inside files.

Dataset Registration is the starting point of working with a dataset. It happens by inserting dataset file naming rules into the hierarchy by ChronosDB administrators by manually editing the XML file. This way of registration is the best practice exploited by proven data servers [76, 18].

Dataset Metadata M (section 2.3) resides on the Gate together with arrays $A.d_i$ and elements of $\forall p \in P$ except $p.A$. It is often in practice that $A.d_i[j] = start + j \times step$, where $j \in [0, |A.d_i|) \subset \mathbb{Z}$, $start, step \in \mathbb{R}$. Hence, only $|A.d_i|$, *start* and *step* values have to be usually stored.

Dataset Files are placed on cluster nodes by manual copying or the `retrieve` ⟨`node`⟩ ⟨`file`⟩* command (download the given list of files via FTP to the specified node).

A file of a raw or regular dataset is always stored entirely on a node unlike in parallel or distributed file systems. A file may be replicated (manually or using commands) on several nodes for fault tolerance and load balancing. A file is a subarray from P^{raw} or P. The ChronosDB array model is designed to be as generic as possible but allowing the establishment of 1:1 mapping of a subarray to a dataset file.

ChronosDB assumes that all dataset files have the same (1) format, (2) set of variables (bands) with the same names, (3) set of attributes, and other relevant characteristics. Also, variables with the same name in distinct files have the same set and order of axes. In practice, data providers (NASA, ECMWF, etc.) already disseminate files satisfying all these criteria. The `validate` command can also verify this.

File Discovery. On startup or the receipt of the `discover` command, workers get the dataset hierarchy from the Gate, scan their local filesystems to discover datasets, create $p.M$, $p.B$, $p.E$, $p.key$ by parsing file names or reading file metadata, and send $\forall p$ (except $p.A$) to the Gate which broadcasts back $A.d_i$ for $\forall i$ if the dataset satisfies raw or regular criteria (see below). Periodic auto-rescan is not yet supported.

Raw Dataset Ingestion. Subarrays in discovered files may have different coordinate reference systems (CRS) [15]. In addition, subarrays may be misaligned despite having the same CRS: $\nexists j : p_1.A.d_i[j] = p_2.A.d_i[j]$ but p_1 and p_2 spatially overlap over the ith axis. This is reported to the user and they must issue the `mosaic` command. Its functionality is parallelized and overlaps with the Oracle Spatial and Graph `SDO_GEOR_AGGR.mosaicSubset` procedure [45].

Using the `mosaic` command, we reprojected the scenes into UTM zone 32N, 30 meters/pixel spatial resolution and aligned them given the reference coordinates 500,000 (x) and 5,500,000 (y) by the nearest-neighbor resampling, fig. 1 (left).

At this stage ChronosDB has two more assumptions about dataset files: all files have the same CRS and alignment.

Regular Dataset Ingestion makes raw dataset files satisfy regular dataset criteria concerning subarray partitioning (section 2.3) using the `retile` command (section 4.1).

ChronosDB created the regular dataset by retiling the 32 Landsat scenes (8.3 GB) prepared by the `mosaic` command in ≈**30 seconds** on the 8-node cluster, fig. 1 (right). SciDB imported the mosaic file for a single band (3.8 GB) in ≈**2 hours** on a powerful server to avoid burning Cloud time.

At this stage ChronosDB adds an additional partitioning assumption about (regular) dataset files. Subarrays of the dataset also become evenly distributed among cluster nodes using the default placement policy, section 4.1.

Array Schema definitions/notations greatly differ among existing array DBMS; there is no well-established practice yet. ChronosDB again follows its command line mainstream.

Tools `ncks` (NCO), `gdalinfo` (GDAL), and others print file metadata and array schema to `stdout`. ChronosDB commands with the same names print array (dataset) schema in plain text. In addition, the commands format their output analogously to the respective tools.

Users avoid learning a new array schema notation and inspect ChronosDB datasets in a way they are accustomed to with only minor changes to the output (e.g. additional information about the retiling parameters used: `Subarray=2048x2048`).

ChronosDB does not yet allows the creation of an array by defining its schema. ChronosDB infers array schema for raw and regular datasets by analyzing their files.

Below is the ChronosDB `gdalinfo` command output for the array of the illustrative dataset "Band №4". Note that `gdalinfo` prints the same output for the raw (after `mosaic`) and regular (after `retile`) illustrative datasets. The `Coordinate System`, `Type`, `Pixel Size` etc. are the same for all dataset files while `Size` and `Corner Coordinates` were automatically inferred. The meaning of the fields in this output is the same as for the well-known `gdalinfo` tool [20].

```
gdalinfo Landsat8.Level_1.SurfaceReflectance.Band4
Driver: GTiff/GeoTIFF
Size is 38673, 24937
Coordinate System is:
PROJCS["WGS 84 / UTM zone 32N", ...skipped... AUTHORITY["EPSG","32632"]]
Origin = (-53110.000000000000000,5878570.000000000000000)
Pixel Size = (30.000000000000000,-30.000000000000000)
Metadata:
  AREA_OR_POINT=Area
Corner Coordinates:
Upper Left  (  -53110.000, 5878570.000) (  0d47'37.30"E, 52d46'20.14"N)
Lower Left  (  -53110.000, 5130460.000) (  1d50'34.16"E, 46d 6'11.15"N)
Upper Right ( 1107080.000, 5878570.000) ( 17d59'48.38"E, 52d42'52.21"N)
Lower Right ( 1107080.000, 5130460.000) ( 16d50'57.11"E, 46d 3'26.67"N)
Center      (  526985.000, 5504515.000) (  9d22'26.95"E, 49d41'33.18"N)
Subarray=2048x2048 Block=2048x1 Type=UInt16, ColorInterp=Gray
  NoData Value=0
```

ChronosDB expects a user to perceive an array as a single object despite the fact that the array actually consists of a variable number of files scattered among cluster nodes.

Metadata Management also happens via ChronosDB commands that mirror the functionality of the respective tools. ChronosDB currently supports the analog of `ncatted` (attribute editor, NCO) to append, create, delete, modify, and overwrite internal file metadata [43]. `ncatted` creates a new dataset since ChronosDB datasets are read-only.

ChronosDB can potentially provide a broader set of similar commands since all dataset files contain the same set of metadata keys and a metadata operation is usually reduced to modifying each file individually.

Location Management commands include `cp` (make a copy of a dataset), `rm` (remove a dataset), `mv` (place a subset of subarrays on a given worker), `retile` (section 4.1).

For example, `cp Landsat8.Level_1.SurfaceReflectance.Band4 $band4` creates a copy of the dataset (Band № 4) residing in RAMD.

3.4 A Life of a Script

We demonstrate how ChronosDB executes scripts and how it is efficient for complex analytic pipelines as follows.

NDVI = (NIR − RED)/(NIR + RED + 1) is a popular estimate of the photosynthetic activity of plants [39]. Soil-Adjusted Vegetation Index SAVI = (NIR−RED)/(NIR+RED+L)×(1+L) aims to minimize soil brightness influences. L is a soil fudge factor varying from 0 to 1 depending on the soil [28]. Note that we may tune L many times to find an appropriate value.

We take NIR and RED arrays with different subarray shapes 2048×2048 and 4096×4096 to avoid their collocation (subarrays from different datasets are collocated if they reside on the same node and their coordinate arrays overlap), to trigger retiling and network exchange. We first perform their 2× interpolation, then join them to compute SAVI, and then create a small 1209×780 "quick outlook" array (64× less than the join output) to visually estimate the result. Note that changing L triggers recomputing of the overall pipeline.

```
gdalwarp -tr 15 15 Landsat8.Level_1.SurfaceReflectance.Band4 $redWarp
gdalwarp -tr 15 15 Landsat8.Level_1.SurfaceReflectance.Band5 $nirWarp
join -A $nirWarp -B $redWarp --jtype inner
     --reduce="(A.astype(float)-B)/(A.astype(float)+B+0.8)*(1+0.8)"
     --type=Float32 --out $SAVI
gdalwarp -tr 960 960 -overwrite $SAVI SAVIoutlook
```

`gdalwarp` accepts the target resolution: 15 leads to the 2× increase of the resolution (recall that the source pixel size is 30 meters), 960 leads to 64× downsampling ($960/15 = 64$).

ChronosDB commands cannot be nested: they have input and output datasets. The output dataset of a command can be specified as the input dataset for the next command. This clear and easy-to-use syntax is not a limitation or performance penalty. ChronosDB does not materialize the whole output of a command before launching the next command.

Script Parsing. The Gate has a built-in command line parser to get the command name and its arguments from a string like "`command arg1 arg2` ...". To integrate a tool named `command` in ChronosDB, a tool driver (Java class) of the same name must be created. The parser instantiates the tool driver named `command` and passes to it the parsed arguments. The driver returns an operation executor depending on the arguments. The executor must implement GET-KEYS, GET-SUBARRAY, and other methods to build an execution plan (see below and section 4). A tool driver may also verify the syntax and the absence of malicious instructions since some options are passed to the OS as is.

Intermediate Datasets are prefixed with $ and reused in subsequent commands. Such datasets are not registered as permanent in the namespace to save time. A VFF is also used for them when possible: if the operation is not an algebraic computation or hyperslabbing and if a tool supports a VFF. For example, we can completely avoid materialization during the interpolation and the merge step of the join.

Execution Plan is an acyclic graph $G = (V, E)$, where $E = \{((\mathbb{D}_i, key_i^j), (\mathbb{D}'_k, key_m)) : \mathbb{D}_i\langle\langle key_i^j\rangle\rangle$ is required to compute $\mathbb{D}'\langle\langle key_m\rangle\rangle\}$. For an array operation, $\mathbb{D}_i$ is an input dataset and $\mathbb{D}'_k$ is the output dataset. The Gate builds a static G for a script (G is immutable at runtime). This makes it easier to reason about the data flow.

Each operation has GET-KEYS and GET-SUBARRAY functions executed on the Gate and workers respectively (section 4). GET-KEYS (1) computes the keys of $\mathbb{D}'_k$ and $\forall p \in \mathbb{D}'_k.P$ except $p.A$ given the metadata of input datasets, and returns its part of the graph G, (2) may assign the destination worker ID to $\mathbb{D}'_k\langle\langle key_m\rangle\rangle.wid$. Step (1) is possible since all operations are formally defined and the Gate locally keeps all necessary metadata (for step (1) the Gate has functions similar to GET-SUBARRAY, not shown in section 4).

GET-SUBARRAY physically produces subarray $\mathbb{D}'_k\langle\langle key_m\rangle\rangle$ given key_m and all necessary metadata. It takes as input all edges $\{(u, v)\} \subseteq E$ such that $v = (\mathbb{D}'_k, key_m)$.

Plan Rewriting capabilities currently consist of inserting a retiling with different parameters to maintain load balance and to control the sizes of subarrays (see below).

Runtime Dataset Consistency. ChronosDB commands treat regular dataset files such that they never violate the assumptions from section 3.3. Each command also verifies its input datasets. For example, all input datasets for the K-way array join (section 4.2) must have the same (1) CRS, (2) set and order of axes, and (3) resolution along each axis.

Vacuum service scans subarrays in the background when ChronosDB is idle and removes those fully consisting of `NA`. This avoids additional costs of testing each subarray during script execution and keeps execution plans static.

Controlling Sizes of Subarrays. Subarrays may grow too large or become too small during their processing. A few large or many small subarrays may also be produced.

ChronosDB uses retiling to merge/split small/large subarrays. We examine the execution plan and find operations after which a subarray size becomes $2\times$ more or less a given threshold. ChronosDB inserts the retiling (taking all its parameters from the dataset except the subarray shape) in the plan after such an operation e.g. after `SAVIoutlook`.

We investigate subarrays for potential retiling only if they are not in a VFF, are going to be materialized, and are not at the border of their array e.g. $\mathbb{D}'\langle\langle -4, 2\rangle\rangle$, fig. 2f.

Task Scheduling. We consider a $v \in V$ to be a task that must be assigned to a worker based on input data locality. If a subarray is placed redundantly, we randomly choose a replica. An output subarray will be moved to the worker $\mathbb{D}'_k\langle\langle key_m\rangle\rangle.wid$ or kept locally if wid was not assigned.

Depth-first search starts from any of the resulting datasets, e.g. `SAVIoutlook`, traverses the reverse of G and determines the order of producing subarrays: $\delta : v \mapsto \Delta$. δ is assigned upon exit, $v \in V$, $\Delta \subset \mathbb{Z}$, $|\Delta| > 1$ if v was visited more than once.

Note that after a hyperslabbing operation some vertices may become unreachable from any vertice of any resulting dataset. This automatically prunes some redundant tasks.

ChronosDB tasks are (1) tiny (each processes one out of hundreds or thousands of subarrays; each RED and NIR dataset has 232 subarrays shaped $\approx 2048\times2048$) and (2) finish at such a high rate (hundreds of milliseconds) that it is reasonable to usually assign the upcoming task based on data locality and not to move the data to another worker. The task to be scheduled may wait a little for the worker with the input data for the task to be free since the time for moving this data is often comparable to the waiting time.

In addition, ChronosDB tasks are of the same workload: they (3) generally have equal volumes of input/output data, and (4) the same operation is applied in each task.

Recent works show that fine-grained tasks alleviate problems related to fairness, stragglers, skew and make it easier to evenly load resources and increase cluster utilization [47].

Experiments suggest (section 5.3) that given task properties (1–4), a static scheduler is usually sufficient when (i) input subarrays (for each operation in the script) are evenly distributed among workers, (ii) stragglers are absent, nodes are homogeneous, and (iii) only one script can run at a time.

ChronosDB ensures (i) by inserting a retiling, see below.

SciDB shuffles chunks among all cluster nodes almost after each operation, section 5.1. This ensures scalability at a high cost but not straggler mitigation or proper multi-tenancy.

Maintaining Load Balance. ChronosDB traverses the execution plan and counts workers holding over $S/n \times \lambda_1$ or less than $S/n/\lambda_1$ subarrays for a dataset $\mathbb{D}$. If it finds more than n/λ_2 such workers, it inserts the retiling with parameters taken from $\mathbb{D}$. The retiling will redistribute subarrays among workers using the default placement policy (section 4.1) keeping subarrays intact. Here λ_1, λ_2 are thresholds, $S = \sum_{\forall p\in\mathbb{D}.P} |p.A|$, and n is the number of workers.

This prevents the majority of subarrays to concentrate on a small set of workers and ensures amortized scalability.

Example Plan. Figure 2e shows the example of the SAVI execution plan built for tiny RED (Band4) and NIR (Band5) arrays with different shapes of subarrays. This triggers the retiling of Band5 during the join. The tuples contain the keys of subarrays, the subscript is the δ value: $key = (k_1, k_2, \dots)_\delta$, e.g. $(0, 1)_9$ refers to Band4$\langle\langle 0, 1\rangle\rangle$.

Plan Execution. The Gate sends to a worker the whole subgraph with all vertices assigned to that worker. A worker keeps all tasks received from the Gate in a local priority queue Q sorted in the ascending order by δ.

Workers execute tasks in a push-based fashion. Once a slot in the local execution pool is free, the worker picks the first task $u \in Q$ for which all input subarrays are locally available and submits u to the pool. The worker materializes the output subarray s_u to its local RAMD or to SSD if the RAMD is full. Let $(u, v) \in E$ and v be assigned to another worker with ID $= w_a$. Then the worker connects to w_a and reports the input subarray s_u for v to be ready. Worker w_a replies to the reporter immediately if $\delta(v) = min(Q)$ or once (1) $\delta(v)$ becomes one of the smallest in Q: $\delta(v) - min(Q) < \Theta_\delta$ and (2) RAMD is at least Θ_{RAMD}% free, where Θ_δ and Θ_{RAMD} are thresholds. When the worker receives the reply, it streams s_u to w_a. If $|\delta(u)| > 1$, the worker keeps s_u until all its derived subarrays are obtained (by this or other workers).

Note that all input subarrays for v may arrive to w_a before v is considered for execution. This enables starting v at once when its turn comes not waiting for the arrival of input data.

A multi-threaded tool occupies that number of slots which are currently free, it does not wait for the 100% free pool to run. The number of threads for such a tool is set to the number of free slots (as the command line parameter).

Fault Tolerance. Suppose subarrays for $V' \subset V$ were not produced. The scheduler is then run to reassign workers for $\forall v \in V'$ and the execution for this subgraph is started.

4. ARRAY OPERATIONS

We now show how to perform a wide variety of distributed operations leveraging elaborate command line tools. Each algorithm has lines highlighted in light gray. These are isolated parts of the algorithms which are delegated to a tool.

At a glance, it may seem there is no room for improvements in some operations and the delegation will not yield a speedup. However, many steps contribute significantly to the elapsed time required to complete even a relatively simple operation. For example, the aggregation includes I/O, byteswapping, broadcasting (e.g., conforming scalars to multi-dimensional variables), collection (e.g., accessing discontinuous hyperslabs in memory), and possibly weighting [89, 90, 88]. Even the averaging runs several times faster when optimized properly [89]. The same is true for highly optimized, industry-standard tools used for other operations [43, 19]. The tools also have a wealth of code for proper metadata maintenance in complex raster file formats: a subarray is readily accessible by any other GIS software.

4.1 Distributed N-d Retiling

ChronosDB uses retiling to process a raw dataset into a regular one, to reprocess (*retile*) a regular dataset with different parameters, for K-way joins (section 4.2). It is also the core of many other vital functions (section 3.4).

The basic idea is to cut each $p \in P$ onto smaller pieces $P' = \{p' : p' \sqsubseteq p\}$, assign each piece a key, and merge all pieces with the same key into a single, new subarray. Each key is associated with a cluster node, all pieces with the same key are gathered and merged on the same node.

Algorithm 1 tiles and retiles sets of both arbitrarily and regularly shaped N-d arrays, accepts an overlap, reference indexes, manages subarrays with keys, fits the ChronosDB data model, is formalized and parallelized.

Algorithm 1 has two phases: cut (split subarrays into pieces) and merge (the pieces) which are launched in turn when the Gate encounters the `retile` command in a script.

Algorithm 1 is illustrated on a 2-d case, fig. 2f. Consider a 2-d array $A(lat, lon)$ in a regular dataset $\mathbb{D} = (A, M, P, S, \rho, r^0)$, where $S = (5, 6)$, $\rho = (0, 0)$, $r^0 = (5, 6)$. Array A has shape 10×18 and consists of 6 subarrays which bodies are separated by thick blue lines. Subarrays may reside on different cluster nodes. Cells are separated by dotted gray lines. Subarrays have key range $k^0_1 \in [-1, 0]$, $k^0_2 \in [-1, 1]$.

Dataset $\mathbb{D}$ is being retiled into $\mathbb{D}' = (A, M, P', S', \rho', \bar{r}^0)$, where $S' = (3, 3)$, $\rho' = (1, 1)$, $\bar{r}^0 = (12, -1)$. New subarray bodies are separated by dashed red lines. In total, 28 new subarrays will be produced with key range $\bar{k}^0_1 \in [-4, -1]$, $\bar{k}^0_2 \in [0, 6]$. For example, the hatched area marks subarray $\mathbb{D}'\langle\langle -3, 2\rangle\rangle = A[2\!:\!6, 4\!:\!8]$ with body $A[3\!:\!5, 5\!:\!7]$ and periphery $A[6, 4\!:\!8]$, $A[2, 4\!:\!8]$, $A[3\!:\!5, 4]$, and $A[3\!:\!5, 8]$. Not all new subarrays will have shape 5×5 ($5 = s'_1 + \rho'_1 \times 2$) and fully cover respective subspaces, e.g. $\mathbb{D}'\langle\langle -1, 6\rangle\rangle = A[8\!:\!9, 16\!:\!17]$.

Reference coordinates r^0_i and $\bar{r}^0_i$ are highlighted in pink on *lat* and *lon* axes. The source and new subarray keys are plotted on axes k^0_i and $\bar{k}^0_i$ respectively.

Function Get-Key calculates the ith key value $\bar{k}^0_i$ of the new subarray which body contains coordinate y_i.

Function Get-Extents takes a set of subarrays and finds out how to cut each of them into pieces (arrays of indexes B' and E', line 26) so cells within each piece belong to the same new subarray. B' and E' are later used by workers to perform the actual cutting of pieces (line 13).

Algorithm 1 Distributed N-d Retiling with an Overlap

```
Input: $\mathbb{D} = (A, M, \dots)$                        ▷ Raw or regular dataset
       $S' = (s'_1, s'_2, \dots, s'_N)$                     ▷ Target shape for new subarrays
       $\rho' = (\rho'_1, \rho'_2, \dots, \rho'_N)$          ▷ Overlap
       $\bar{r}^0 = (\bar{r}^0_1, \bar{r}^0_2, \dots, \bar{r}^0_N)$   ▷ Reference indexes, $\bar{r}^0 \subset \mathbb{Z}^N$
       function Policy                                      ▷ Piece placement policy
Output: $\mathbb{D}' = (A, M, P', S', \rho', \bar{r}^0)$
Require: $s'_i \in [\Theta^b_{axis}, \Theta^e_{axis}] \subset \mathbb{Z}$, $s'_i > 0$, $\prod_{i=1}^N s'_i \leqslant \Theta_{shape}$
 1: function Retile($\mathbb{D}, S', \rho', \bar{r}^0$, Policy)  ▷ is executed on the Gate
 2:     Initiate two-phase retiling: (1) [Cut], (2) [Merge]
 3:     return $\mathbb{D}' = (A, M, P', S', \rho', \bar{r}^0)$
 4: function Get-Keys [Cut]($\mathbb{D}$)                      ▷ the Gate has
 5:     $(\mathbb{C}, \mathbb{K}) \leftarrow$ Get-Extents($\mathbb{D}.P$)        ▷ $\forall p \in \mathbb{D}.P$ except $p.A$
 6:     return $\{((\mathbb{D}, key_1), (\mathbb{D}_{cut}, (key_1, key_2))) : (key_1, key_2) \in \mathbb{K}\}$
 7: function Get-Keys [Merge]($\mathbb{D}_{cut}$)
 8:     return $\{((\mathbb{D}_{cut}, (k_1, k_2)), (\mathbb{D}', k_2)) :$ Policy$(\mathbb{D}', k_2) \neq$ SKIP$\wedge$
 9:     for $\forall (k_1, k_2) \in keys(\mathbb{D}_{cut}), \mathbb{D}'\langle\langle k_2\rangle\rangle.wid \leftarrow$ Policy$(\mathbb{D}', k_2)\}$
10: function Get-Subarray [Cut]($\{((\mathbb{D}, k_1), (\mathbb{D}_{cut}, (k_1, k_2)))\}$)
11:     $(\mathbb{C}, \mathbb{K}) \leftarrow$ Get-Extents($\mathbb{D}.P$)  ▷ a worker has all local $p \in \mathbb{D}.P$
12:     find $(p, key, B', E') \in \mathbb{C} : p.key = k_1 \wedge key = k_2$
13:     $\mathbb{D}_{cut}\langle\langle k_1, k_2\rangle\rangle.A \leftarrow p.A[B'[0]\!:\!E'[0], \dots, B'[N-1]\!:\!E'[N-1]]$
14: function Get-Subarray [Merge]($\{((\mathbb{D}_{cut}, (k_1, k_2)), (\mathbb{D}', k_2))\}$)
15:     $\mathbb{D}'\langle\langle k_2\rangle\rangle \leftarrow$ merge $\{\mathbb{D}_{cut}\langle\langle k_1, k_2\rangle\rangle\}$ for $\forall k_1$  ▷ DELEGATION
16: function Get-Key($y_i$)
17:     return $(y_i \geqslant \bar{r}^0_i)\,?\,(y_i - \bar{r}^0_i)$ div $s'_i : -1 - ((\bar{r}^0_i - 1 - y_i)$ div $s'_i)$
18: function Get-Extents($P$)                     ▷ $\mathbb{C}$: cut extents from $\forall p \in P$
19:     $\mathbb{C} \leftarrow \mathbb{K} \leftarrow \{\}$, $k^b_i \leftarrow$ Get-Key$(0)$,  $k^e_i \leftarrow$ Get-Key$(|A.d_i| - 1)$
20:     for each $p \in P$ do                    ▷ $b_i = p.B[i-1]$, $e_i = p.E[i-1]$
21:         $x^b_i \leftarrow$ Get-Key$(b_i)$,  $x^e_i \leftarrow$ Get-Key$(e_i)$
22:         for each $\bar{k}_i \in [max(k^b_i, x^b_i - 1), min(k^e_i, x^e_i + 1)]$ do
23:             $b'_i \leftarrow max(\bar{r}^0_i + \bar{k}_i \times s'_i - \rho'_i - b_i, 0)$
24:             $e'_i \leftarrow min(\bar{r}^0_i + (\bar{k}_i + 1) \times s'_i - 1 + \rho'_i - b_i, e_i - b_i)$
25:             if $e'_i < 0 \vee b'_i > e_i - b_i$ then *continue*
26:             $B'\langle N\rangle = \{b'_1, \dots, b'_N\}$,  $E'\langle N\rangle = \{e'_1, \dots, e'_N\}$
27:             $key_1 \leftarrow p.key$, $key_2 \leftarrow (\bar{k}_1, \dots, \bar{k}_N)$
28:             $\mathbb{C} \leftarrow \mathbb{C} \cup \{(p, key_2, B', E')\}$,  $\mathbb{K} \leftarrow \mathbb{K} \cup \{(key_1, key_2)\}$
29:     return $(\mathbb{C}, \mathbb{K})$
```

We find the key range of the new subarrays into which A (line 19) and p (line 21) are to be cut. Line 22 steps a key back (-1) and a key forward ($+1$) to ensure cutting the periphery of the new subarrays whose bodies do not overlap with p. Here $k^b = (-4, 0)$ and $k^e = (-1, 6)$, fig. 2f. Lines 22–28 iterate over the latter key range limited by k^b_i and k^e_i to avoid generating subarrays without bodies: $A[0, 1\!:\!5]$ will be not cut and $\mathbb{D}'\langle\langle -5, 1\rangle\rangle$ will be not produced.

Lines 23 and 24 find indexes of $gch(p, \mathbb{D}'\langle\langle \bar{k}_1, \bar{k}_2, \dots, \bar{k}_N\rangle\rangle)$. For example, source subarray $A[5\!:\!9, 0\!:\!5]$ will be cut into 9 pieces, one of them is $A[5\!:\!6, 4\!:\!5]$ belonging to the new dashed subarray, fig. 2f. Here $x^b = (-3, 0)$ and $x^e = (-1, 2)$.

The retiling takes Policy function as an argument to customize the placement of each new subarray, line 9. Two options are currently supported; for all pieces with a given key: do not cut them (`SKIP`) or send them to worker *wid* (may be equal to the source worker id). Note that if a key is skipped, no subarray will be produced with this key.

Default-Policy function (not shown in algorithm 1) is taken by default for retiling. It returns `SKIP` if a subarray with a given key has no body. It sorts other subarrays by keys and assigns the first subarrays with the total size of $\approx S/n$ to the first worker, the next portion of subarrays to the second worker and so on. Here S is the total size of subarrays and n is the number of workers. Given many tiny and equal-sized tasks, this leads to an even load distribution among workers without complex bin packing algorithms.

ChronosDB regular subarrays are similar to RasDaMan tiles or SciDB chunks serving as I/O units. RasDaMan

Algorithm 2 Distributed, File Based K-Way Array Join

```
Input: $\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_K$            ▷ Datasets
       $S, \rho, r^0$                                  ▷ Parameters used for cooking $\mathbb{D}_1$
       $\odot \in \{\textsc{inner}, \textsc{outer}\}$            ▷ Join Type
       $\kappa(p_1, p_2, \ldots, p_K)$                  ▷ K-ary operation
Output: $\mathbb{D}_{\bowtie} = (A_{\bowtie}, M, P_{\bowtie}, S, \rho, r^{(0)})$
Require: $\mathbb{D}_1$ must be regular, it is a "reference" dataset
 1: function Get-Keys
 2:     $r_i^{(0)} \leftarrow r_i^0 + z_i,\ z_i : \mathbb{D}_1.A.d_i[0] = d_i^{\bowtie}[z_i],\ \ \mathbb{P} \leftarrow \mathbb{D}_1.P$
 3:     for each $k \in [2, K] \subset \mathbb{Z}$ do
 4:         $\bar{r}_i^{(0)} \leftarrow r_i^{(0)} - z_i^0,\ z_i^0 : \mathbb{D}_k.A.d_i[0] = d_i^{\bowtie}[z_i^0]$
 5:         $\mathbb{D}'_k \leftarrow$ Retile($\mathbb{D}_k, S, \rho, \bar{r}^{(0)}$, K-Join-Policy), $\mathbb{P} \leftarrow \mathbb{P} \cup \mathbb{D}'_k.P$
 6:     $\mathbb{K}_j \leftarrow keys(\mathbb{D}'_j),\ \circ \leftarrow (\odot = \textsc{inner})\,?\cap : \cup,\ \mathbb{K} \leftarrow \mathbb{K}_1 \circ \mathbb{K}_2 \circ \cdots \circ \mathbb{K}_K$
 7:     compute $P_{\bowtie}$, $\mathbb{D}_{\bowtie} \leftarrow (A_{\bowtie}, M, P_{\bowtie}, S, \rho, r^{(0)})$
 8:     return $\{((\mathbb{D}'_j, k), (\mathbb{D}_{\bowtie}, k)) : j \in [1, K] \wedge k \in \mathbb{K} \cap keys(\mathbb{D}'_j)\}$
 9: function K-Join-Policy($\mathbb{D}'_k, key$)
10:     if $\exists wid \in \{p.wid : p \in \mathbb{P} \wedge p.key = key\}$ then return $wid$
11:     return $(\odot = \textsc{inner})$ ? SKIP : Default-Policy($\mathbb{D}'_k, key$)
12: function Get-Subarray($edges = \{((\mathbb{D}'_j, key), (\mathbb{D}_{\bowtie}, key))\}$)
13:     for $j \in [1, K]$ do
14:         $p_j \leftarrow [\exists((\mathbb{D}'_j, key), (\mathbb{D}_{\bowtie}, key)) \in edges]\ ?\ \mathbb{D}'_j\langle\langle key\rangle\rangle$ : NA
15:     return $\mathbb{D}_{\bowtie}\langle\langle key\rangle\rangle \leftarrow \kappa'(p_1, p_2, \ldots, p_K)$        ▷ DELEGATION
```

and SciDB do not further partition cells inside a tile/chunk. However, ChronosDB subarrays are stored in sophisticated raster file formats the majority of which support chunking (section 4.7). This enables additional partitioning level of cells and more flexibility in adapting to dynamic workloads.

4.2 Distributed K-Way Array Join

Map algebra is an analysis language loosely based on the concepts presented in [80]. It is widely used in the industry [87]. Map algebra includes algebraic operations on georeferenced N-d arrays. Many K-ary array operations implicitly require the prior K-way array join if $K > 1$.

ChronosDB supports two join types. INNER join is useful for algebraic computations when any operation on a set of cells must return NA if at least one of the cell values is missing. OUTER join is useful for a compositing operation when at least one non-missing value contributes to the resulting cell, e.g. a K-day cloud-free composite of satellite imagery.

The K-way join $\bowtie$: $\odot, \kappa, A_1, A_2, \ldots, A_K \mapsto A_{\bowtie}$ takes as input a join type $\odot \in \{\textsc{inner}, \textsc{outer}\}$, a mapping $\kappa : \mathbb{T}_1, \mathbb{T}_2, \ldots, \mathbb{T}_K \mapsto \mathbb{T}_{\bowtie}$, and N-d arrays $A_j(d_1^j, d_2^j, \ldots, d_N^j) : \mathbb{T}_j$, $j \in [1, K]$ such that $\exists d_i^{\bowtie} : d_i^j \sqsubseteq d_i^{\bowtie}$ for $\forall i, j$.

The K-way join yields the N-d array $A_{\bowtie}(d_1^{\bowtie}, d_2^{\bowtie}, \ldots, d_N^{\bowtie}) : \mathbb{T}_{\bowtie}$ such that $A_{\bowtie}[x_1, x_2, \ldots, x_N] = \kappa'(a_1, a_2, \ldots, a_K)$, where $x_i \in [0, |d_i^{\bowtie}|)$, $a_j = A_j[y_1^j, y_2^j, \ldots, y_N^j]$ if $\exists y_i^j : d_i^j[y_i^j] = d_i^{\bowtie}[x_i]$; $a_j = \texttt{NA}$ otherwise.

Operation κ' formally defines the difference between the two join types. If $a_j = \texttt{NA}$ for $\forall j$, κ' returns NA regardless of the $\odot$ value. If $\odot = \textsc{inner}$ and $\exists j : a_j = \texttt{NA}$, κ' returns NA. Otherwise κ' returns $\kappa(a_1, a_2, \ldots, a_K)$.

Unlike SciDB join [67], ChronosDB join (1) allows joining more than two arrays at a time, (2) provides OUTER and INNER join types, (3) accepts a reducer to derive an array from the joined arrays (step 3 often follows the join in practice).

ChronosDB uses a retiling-based approach for the K-way array join. The main goal of the join algorithm is to prepare input arrays such that κ could be delegated to an external tool. Most tools work on a single machine and refuse to process arrays not covering exactly the same N-d subspace.

We (1) retile all input subarrays with the parameters taken from the first dataset $\mathbb{D}_1$, (2) assign the same key to all new subarrays (regardless of the dataset) whose bodies are in the same N-d subspace defined by $d_i^{\bowtie}$, S, ρ, and $r^{(0)}$ (lines 2 and 4 adjust reference indexes respectively), (3) collect all subarrays with the same key from all datasets on one of the workers, (4) delegate the calculation of κ on the subarrays with the same key to a tool: `ncap2` (NCO) or `gdal_calc.py` (GDAL) depending on the format.

Get-Keys iteratively calls Retile on each input dataset with adjusted reference indexes and the custom piece placement policy. A piece with a given key is assigned to a worker that already has a subarray with the same key from dataset $\mathbb{D}_1$ or a newly retiled dataset. INNER join discards subarrays with keys absent in at least one of the datasets. Otherwise, Default-Policy chooses a worker for a new subarray.

Before delegating κ we must prepare p_j such that $\forall p_j$ have the same extents. If $\odot = \textsc{inner}$, find $d'_i = gch(p_1.A.d_i, \ldots, p_K.A.d_i)$ and cut each p_j respectively. If $\odot = \textsc{outer}$, find $d'_i : p_j.A.d_i \sqsubseteq d'_i \wedge |d'_i| \leqslant s_i + \rho_i \times 2$ for $\forall i, j$ and extend each p_j respectively filling newly emerged cells with NA.

The K-way join algorithm is amenable to modifications. For example, to multiply 2-d arrays A and B, where $S_A = (a_1, a_2)$, $S_B = (b_1, b_2)$ we can (1) retile subarrays of B to shape (a_2, a_1), (2) use the policy function which just reverses the $key = (k_1, k_2)$ of a newly cut piece (from B) to $key' = (k_2, k_1)$ and seeks the subarray of A with key'. This idea is applicable to similar operations where the column blocks are joined with row blocks. It is also straightforward to extend the idea for similar K-ary operations on N-d arrays.

4.3 Aggregation

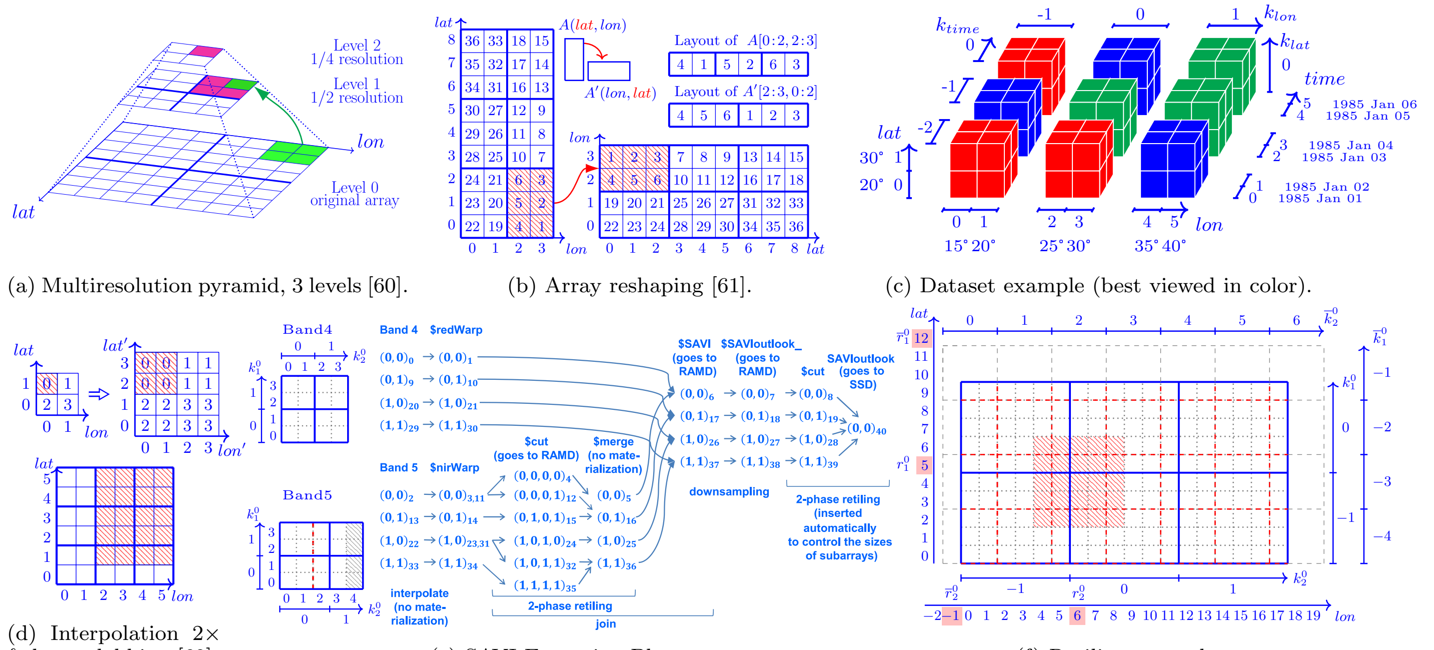
The aggregate of an N-d array $A(d_1, d_2, \ldots, d_N) : \mathbb{T}$ over axis d_1 is the $(N-1)$-d array $A_{aggr}(d_2, \ldots, d_N) : \mathbb{T}_{aggr}$ such that $A_{aggr}[x_2, \ldots, x_N] = f_{aggr}(cells(A[0 : |d_1| - 1, x_2, \ldots, x_N]))$, where $\forall x_i \in [0, |d_i|)$, $cells : A' \mapsto T$ is a multiset of all cell values of an array $A' \sqsubseteq A$, $f_{aggr} : T \mapsto w \in \mathbb{T}_{aggr}$ is an aggregation function.

Algorithm 3 Two-Phase Distributed Array Aggregation

```
Input: $\mathbb{D}, f_{aggr}$     Output: $\mathbb{D}_{aggr}$     Require: $N > 1$
1: function Get-Keys [Phase1]
2:     return $\{((\mathbb{D}, key), (\mathbb{D}_L, key)) : key \in keys(\mathbb{D})\}$
3: function Get-Subarray [Phase1]($\xi = \{((\mathbb{D}, k), (\mathbb{D}_L, k))\}$)
4:     return $\mathbb{D}_L\langle\langle k\rangle\rangle \leftarrow f_{aggr}(\mathbb{D}\langle\langle k\rangle\rangle)$        ▷ $|\xi| = 1$, DELEGATION
5: function Get-Keys [Phase2]
6:     return $\{((\mathbb{D}_L, k), (\mathbb{D}_{aggr}, k_a)) : k \in keys(\mathbb{D}_L) \wedge k_a =$
7:     $k[1 : N-1], \mathbb{D}_{aggr}\langle\langle k_a\rangle\rangle.wid \leftarrow$ Default-Policy($\mathbb{D}_{aggr}, k_a$)$\}$
8: function Get-Subarray [Phase2]($\xi = \{((\mathbb{D}_L, k), (\mathbb{D}_{aggr}, k_a))\}$)
9:     return $\mathbb{D}_{aggr}\langle\langle k_a\rangle\rangle \leftarrow f_{aggr}$(all $\mathbb{D}_L$ from $\xi$)  ▷ DELEGATION
```

Due to space constraints, we give algorithm 3 for aggregating subarrays with the same d_j for the same k_a (phase 2), where $j \in [2, N]$. The aggregation is delegated to **ncra** for NetCDF files (lines 4 and 9), where $f_{aggr} \in \{max, min, sum\}$. The calculation of *average* is reduced to calculating the *sum* and dividing each cell of the resulting array on $|A.d_1|$.

Algorithm 3 has two phases similar to the retiling. Consider the dataset described in section 2.3, fig. 2c. The first phase locally aggregates 3-d subarrays over the first axis to obtain interim 2-d aggregates e.g. $\mathbb{D}\langle\langle 0, 0, 0\rangle\rangle \mapsto \mathbb{D}_L\langle\langle 0, 0, 0\rangle\rangle$ (new keys retain the same dimensionality). To compute the final result, the second phase collects all 2-d subarrays of D_L contributing to the given final 2-d aggregate on the node specified by Default-Policy, section 4.1. For example, $\mathbb{D}_L\langle\langle 0, 0, 0\rangle\rangle$, $\mathbb{D}_L\langle\langle -1, 0, 0\rangle\rangle$, $\mathbb{D}_L\langle\langle -2, 0, 0\rangle\rangle \mapsto \mathbb{D}_{aggr}\langle\langle 0, 0\rangle\rangle$.



(a) Multiresolution pyramid, 3 levels [60]. (b) Array reshaping [61]. (c) Dataset example (best viewed in color). (d) Interpolation 2× & hyperslabbing [60]. (e) SAVI Execution Plan. (f) Retiling example.

Figure 2: Illustration of array processing operations.

4.4 Resampling

Multiresolution Pyramid illustrates **downsampling**. Large arrays are interactively visualized depending on the zoom level. Usually several zoom levels are defined, e.g. $Z = \{0, 1, \ldots, 16\}$. Usually at zoom level $z \in Z$ an array is displayed with $2^z\times$ less resolution than the original. A multiresolution pyramid is the stack of arrays for all zoom levels, fig. 2a. The display of downsampled arrays for coarser scales significantly reduces network traffic and system load making this functionality vital for an array DBMS.

Formally, given a 2-d array $A(d_1, d_2) : \mathbb{T}$, its coarser version is the array $A'(d'_1, d'_2) : \mathbb{T}'$ such that $d'_i\langle\lceil l_i\rceil\rangle = \{d_i[0], d_i[2], \ldots, d_i[\lfloor l_i\rfloor \times 2]\}$, where $l_i = |d_i|/2$ and $A'[x'_1, x'_2] = avg(\{A[x_1, x_2] : x_i \text{ div } 2 = x'_i \wedge A[x_1, x_2] \neq \texttt{NA}\})$ for $\forall x_i$, avg is the average and $avg(\varnothing) = \texttt{NA}$. To shorten the formulas, section 4.4 assumes that d_i stores the smallest border cell coordinates and `gdalwarp` starts the processing from $d_i[0]$. ChronosDB delegates the resampling to `gdalwarp`.

Numerous downsampling techniques exist [57]. However, SciDB does not support anything more advanced. In contrast, a large number of tools with plethora of options exist specifically targeted at creating coarser array versions.

Interpolation is a core operation with many applications [57]. It estimates a value at a coordinate $(y_1, \ldots, y_N)$ for an array $A(d_1, \ldots, d_N)$, where $y_i \in (d_i[j], d_i[j+1]) \subset \mathbb{T}_i$, and $j \in [0, |d_i| - 1) \subset \mathbb{Z}$. For example, array resolution can be doubled by interpolation when $y_i = (d_i[j] + d_i[j+1])/2$.

As in downsampling, numerous interpolation techniques exist [57]. The most basic one is nearest neighbor (NN): an unknown cell value is obtained by copying the value from the nearest cell with a known value. SciDB `xgrid` operator mimics NN interpolation [65]: it increases the length of the input array dimensions by an integer scale replicating the original cell values, fig. 2d. This is almost equivalent to NN since a generic interpolation works with an arbitrary scale.

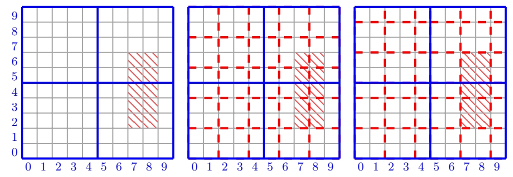
Algorithm 4 Distributed, In Situ Array Hyperslabbing

```
Input: $\mathbb{D} = (A, \ldots)$ ▷ Raw or regular dataset and hyperslab indexes
    $B\langle N\rangle$ : int $= \{b_1, \ldots, b_N\}$, $E\langle N\rangle$ : int $= \{e_1, \ldots, e_N\}$
Output: $\mathbb{D}' = (A', \ldots)$: $A'(d'_1, \ldots, d'_N) = A[b_1 : e_1, \ldots, b_N : e_N]$
Require: $0 \leqslant b_i \leqslant e_i < |A.d_i|$      ▷ result is not empty
 1: function GET-KEYS
 2:     $A'.d'_i \leftarrow A.d_i[b_i : e_i]$
 3:     return $\{((\mathbb{D}, k), (\mathbb{D}', k)) : \forall k \in keys(\mathbb{D}) \wedge \text{GET-CUT}(\mathbb{D}\langle\langle k\rangle\rangle) \neq 0\}$
 4: function GET-CUT($p \in \mathbb{D}.P$)
 5:     $m_i \leftarrow p.A.d_i$, $l_i \leftarrow |m_i| - 1$, $m'_i \leftarrow A'.d'_i$, $l'_i \leftarrow |m'_i| - 1$
 6:     $\omega \leftarrow 0$, $\alpha_i \leftarrow max(m_i[0], m'_i[0])$, $\beta_i \leftarrow min(m_i[l_i], m'_i[l'_i])$
 7:     if $\alpha_i \leqslant \beta_i$ for $\forall i$ then      ▷ $p.A$ and $A'$ overlap
 8:         if $\alpha_i = m_i[0] \wedge \beta_i = m_i[l_i]$ for $\forall i$ then $\omega \leftarrow 1$ ▷ $p.A \sqsubseteq A'$
 9:         else find $b'_i, e'_i : m_i[b'_i] = \alpha_i \wedge m_i[e'_i] = \beta_i$ ▷ cut $gch(p.A, A')$
10:             $\omega \leftarrow (B' = \{b'_1, b'_2, \ldots, b'_N\}, E' = \{e'_1, e'_2, \ldots, e'_N\})$
11:     return $\omega$
12: function GET-SUBARRAY($\xi = \{((\mathbb{D}, k), (\mathbb{D}', k))\}$)      ▷ $|\xi| = 1$
13:     if GET-CUT($\mathbb{D}\langle\langle k\rangle\rangle$) $= 1$ then return copy of $\mathbb{D}\langle\langle k\rangle\rangle$
14:     $(B', E') \leftarrow$ GET-CUT($\mathbb{D}\langle\langle k\rangle\rangle$), $b'_i \leftarrow B'[i-1]$, $e'_i \leftarrow E'[i-1]$
15:     $\mathbb{D}'\langle\langle k\rangle\rangle.A \leftarrow \mathbb{D}\langle\langle k\rangle\rangle.A[b'_1 : e'_1, \ldots, b'_N : e'_N]$   ▷ DELEGATION
```

4.5 Hyperslabbing

Hyperslabbing is an extraction of a hyperslab from an array. Consider a 2-d array $A(lat, lon)$ consisting of 9 subarrays separated by thick lines, fig. 2d. The hatched area marks the hyperslab $A' = A[1:5, 2:5]$. Hyperslabbing an array is reduced to hyperslabbing the respective subarrays as follows. Some subarrays do not overlap with A' and are filtered out, e.g. $A[0:1, 0:1]$. Others are entirely inside A' and migrate to the resulting dataset as is, e.g., $A[4:5, 4:5]$.

We hyperslab only the remaining subarrays to complete the operation. A subarray is entirely located on a node in a single file and its hyperslabbing is delegated to a tool: `gdal_translate` (GDAL) or `ncks` (NCO) depending on the file format (algorithm 4, line 15). Most tools support raster file subsetting on a single machine. ChronosDB scales out the tools by orchestrating their massively parallel execution.



(a) No chunking (b) Exact 2×2 chunking (c) Inexact 2×2 chunking

Figure 3: Array chunking.

4.6 Reshaping

The *reshaping* operation $\Psi : A, \pi \mapsto A'$ takes as input an N-d array $A(d_1, \dots, d_N) : \mathbb{T}$ and the permutation mapping $\pi : i \mapsto j$, where $i, j \in [1, N] \subset \mathbb{Z}$, $\pi(i) \neq \pi(j)$ for $i \neq j$, and $\bigcup_i \{\pi(i)\} = [1, N]$. The reshaping operation outputs the N-d array $A'(d_{\pi(1)}, \dots, d_{\pi(N)}) : \mathbb{T}$ such that $A[x_1, \dots, x_N] = A'[x_{\pi(1)}, \dots, x_{\pi(N)}]$, where $x_i \in [0, |d_i|) \subset \mathbb{Z}$ for all i.

Reshaping helps to achieve the fastest hyperslabbing along a dimension e.g. reading a time series $A[x_1, x_2, 0{:}|time| - 1]$ of a 3-d array $A(lat, lon, time)$ vs. $A(time, lat, lon)$.

It is possible to reshape an array by reshaping its subarrays separately. Consider an array $A(lat, lon)$, shape 9×4, and its reshaped version $A'(lon, lat)$, shape 4×9, thick lines separate the subarrays, fig. 2b. Hatched cells of A and A' mark subarrays $A[0{:}2, 2{:}3]$ and $A'[2{:}3, 0{:}2]$. Hence, $A[b_1 : e_1, \dots, b_N : e_N]$ is reshaped to $A'[b_{\pi(1)} : e_{\pi(1)}, \dots, b_{\pi(N)} : e_{\pi(N)}]$: reshaping does not move cells between distinct subarrays.

Reshaping alters the array storage layout according to the given order of dimensions. Two strips (1-d arrays) show storage layouts of $A[0 : 2, 2 : 3]$ and $A'[2 : 3, 0 : 2]$, fig. 2b. Usually the last dimension of an N-d array varies the fastest: cells along the Nth dimension are sequential in the memory. A cell $A[x_1, x_2]$ precedes the cell $A[x_1, x_2 + 1]$. Thick lines on the strips separate cells with distinct x_1 indexes. Reshaping is delegated to `ncpdq` (NCO) for NetCDF format.

4.7 Chunking

Chunking partitions an array into a set of I/O units called chunks (fig. 3, thick blue and dashed red lines separate subarrays and chunks respectively). Chunking is an approach to substantially accelerate array I/O.

Consider reading a 5×2 slice from a 2-d array, fig. 3. For the row-major storage layout, a possible way is to spend 5 I/O requests to read 5 portions sized 1×2, fig. 3(a). For a compressed array, a larger part of it might have to be read and unpacked to get the slice. However, only chunks storing the required data are read from a chunked array, fig. 3(b,c).

The exact chunking reorganizes data such that the cells with indexes $(x_1, \dots, x_N)$ and $(y_1, \dots, y_N)$ belong to the same chunk iff x_i div $c_i = y_i$ div c_i for $\forall i$, fig. 3(b).

ChronosDB performs exact chunking of subarrays without moving cells between them, fig. 3(c). We call this technique "inexact chunking". Its benefits are: (1) it reaches the goal of chunking (mainly I/O speedup, this is experimentally proven in section 5), (2) it leverages sophisticated chunking techniques of raster file formats, (3) it can be performed in parallel on each subarray, (4) it has a strong practical motivation and is more appropriate for data sharing.

In practice, for a chunk shape $c_1 \times c_2 \times \cdots \times c_N$ and an array $A(d_1, d_2, \dots, d_N)$ the condition $c_i \ll |A.d_i|$ usually holds. This translates to $c_i \ll |p.A.d_i|$ for $\forall p \in P$ given the proper retiling parameters, section 4.1. Real-world datasets are usually already split into files containing subarrays meeting these criteria, e.g. in climate modeling it is common to have files storing yearly or monthly data, section 5.2.

Recall that ChronosDB subarrays are directly accessible by a user and any other software as ordinary files. In the case of yearly files, it is inconsistent to have an exactly chunked file named `2015.*` and supposed to store data for year 2015 but with extra grids from 2014 or 2016 year.

Inexact chunking produces a negligible fraction of chunks with smaller shapes e.g. $A[4, 7{:}8]$, fig. 3(c). This does not necessarily imply a slower I/O performance. For certain access patterns, inexact chunking may yield a faster I/O than the exact one: $A[2{:}6, 7{:}8]$ requires reading 6 (or even 8) vs. 3 chunks from the exactly and inexactly chunked arrays respectively, fig. 3(b, c). Note that if $|A.d_i| \bmod c_i \neq 0$ then the exact chunking also produces some smaller chunks.

Chunk shape is one of the crucial I/O performance parameters for an array [16]. An appropriate chunk shape depends on data characteristics and workload. Generally no single chunk shape is optimal for all access patterns. It is typically not obvious a priori what chunk shape is good in a given case: chunk shape is often tuned experimentally. An array DBMS must be able to quickly alter chunk shape in order to support the tuning and to adapt to dynamic workloads.

ChronosDB leverages `ncks` (NCO) and `gdal_translate` (GDAL) to chunk NetCDF and GeoTIFF files respectively.

5. PERFORMANCE EVALUATION

5.1 Experimental Setup

Hardware and software. We used Microsoft Azure DS2 V2 virtual machines (VMs) with 2 CPU cores (Intel Xeon E5-2673 v3, 2.4 GHz), 7 GB RAM, 64 GB local SSD, max 4 virtual data disks and 6400 IOPS, Ubuntu Linux 14.04 LTS (it is the latest version supported by SciDB 16.9).

SciDB 16.9 is the latest version as of 12/2017. It is mostly written in C++. Parameters used: 0 redundancy, 2 SciDB instances (SIs) per VM, SIs can allocate an unlimited amount of memory, other settings are default [66]. ChronosDB is 100% on Java, ran one worker per node, OracleJDK 1.8.0_111 64 bit, max heap and RAMD are sized 2 GB each. GDAL v2.2.3 and NCO v4.6.7 are from the Conda repository [13].

It is recommended to run 1 SI per 2 CPU cores and 8 GB or more RAM [68, 69]. However, we ran 2 SIs per VM since 1 SI per VM was up to 2× slower for all queries.

We selected DS2 V2 VMs since (1) this hardware was the closest to the minimal required [69] at the time of the experiments, and (2) any experimental array fully fits the RAM of a single VM. DS2 V2 VMs are optimal for these data in terms of cost and performance. A larger VM is not cost effective. We checked most queries on the newer D2S V3 VMs (8 GB RAM) and observed no significant runtime difference compared to DS2 V2.

Shuffling. ChronosDB subarrays and SciDB chunks of the input arrays are distributed uniformly among cluster nodes. Most SciDB operators shuffle output chunks between cluster nodes using a hash function. It assigns a chunk to a SciDB instance via a hash over the chunk position minus array origin, divided by chunk size [64, 63]. This formula may collocate output and input chunks leading to no exchange of chunks over the network, e.g. for `xgrid` operator [64].

Arrows ⇄ in fig. 4 indicate that SciDB actually shuffles the resulting chunks while ⤫ indicates no shuffling or its negligible runtime, e.g. for tiny results in the aggregation. In the former case, ChronosDB runtime also includes shuffling of its output subarrays while in the latter it does not.

For experimental purposes, ChronosDB shuffling makes the jth worker ($j \in [1, n]$) send its output subarrays to the $(n + 1 - j)$th worker, where n is the number of workers. Unlike SciDB, this always forces a complete exchange of all resulting data when n is even. Sometimes this only adds up to 1-2 sec. of overhead due to the fast network, RAMD, zero-copy [77], and the shuffling being parallel to the processing.

Results materialization to SSD is mandatory for many query types since we study real-world use cases. Chunking accelerates many subsequent queries. It is primarily a separate query, not part of a query. We get a new chunked array, save it to SSD, and reuse it many times. Similarly, a pyramid is built in advance to avoid delays when it is served many times via a visualization system (e.g. interactively explore a query output). Reshaping, in particular, accelerates multiple hyperslabbing queries along an axis. Many other queries pursue similar goals. We give many charts (marked by RAM) when SciDB discards its output (`consume` operator) and ChronosDB saves its output to RAMD. ChronosDB is still much faster than SciDB. This translates to an order of magnitude speedup in complex analytic pipelines, e.g. SAVI.

Cold and hot query runs were evaluated: a query is executed for the first and the second time respectively. We drop `pagecache`, `dentries` and `inodes` before each cold query to prevent data caching. ChronosDB benefits from native OS caching and is much faster during hot runs. This is particularly useful for continuous experiments with the same data occurring quite often (e.g. tuning certain parameters, section 4.7). There is no significant runtime difference between cold and hot SciDB runs. Superlinear acceleration in some cases may be due to exceeding the I/O limits leading to a slowdown for smaller clusters compared to larger ones.

It is impossible to import large data volumes into SciDB in a reasonable time frame. However, the datasets turned out to be sufficient for representative results.

It took us about 600+ lines of Bash code for SciDB and about 450+ lines of ChronosDB script to run all operations with different chunk shapes and modes (RAM/SSD).

5.2 Climate Reanalysis Data

Eastward (u-wind) and northward (v-wind) wind speeds at 10 meters above surface between 1979–2010 (32 years) from NCEP/DOE AMIP-II Reanalysis (R2) are taken [41]. These are 6-hourly forecast data (4-times daily values at 00:00, 06:00, 12:00, and 18.00), a time series of 94 × 192 Gaussian grids, NetCDF3 format, ≈3 GB. The data already satisfy regular dataset criteria and need not to be retiled. They result in two 3-d arrays shaped 46752 × 94 × 192. All queries except the wind speed were ran on the u-wind array.

We developed a Java program to convert NetCDF ↦ CSV to feed the latter to SciDB. To date, this is the only way to import a NetCDF file into SciDB. The import took over 45 hours on a powerful server to avoid wasting Cloud time.

An appropriate chunk shape must be found for a SciDB array to get the fastest performance. It is practical to import a single grid for a single time step per iteration. This justifies setting the initial chunk shape to A, table 1 (numbers are for the u-wind array). Shape E coincides with the shapes of ChronosDB subarrays except for leap years. SciDB failed to reshape the array with chunk shape E. For ChronosDB and the u-wind array, only hyperslabbing benefits from chunking since it is more apparent for much bigger data: there is no need to tune chunk shape for ChronosDB in this case.

Table 1: SciDB runtime[1] vs. chunk shape[2], 8 nodes

Operation	A	**B**	C	D	E
Average	63.2	**16.7**	45.2	54.4	17.2
Reshape	210.9	**165.0**	3974.4	1145.2	—
$[, 0:20, 0:20]^3$	138.0	**5.4**	92.8	22.8	25.2
$[, 0, 0]^3$	104.2	1.5	11.3	**0.6**	15.8

[1] seconds, [2]chunk shapes – A: 1 × 94 × 192, B: 100 × 20 × 16, C: 10 × 10 × 8, D: 730 × 2 × 2, E: 1460 × 94 × 192, [3]hyperslabbing

The SciDB aggregation performance is the fastest and approximately the same for arrays with chunk shapes B and E, table 1. It is about 4× slower than the ChronosDB aggregation, fig. 4d. For chunk shape E only 33 chunks at most must be read, aggregated, and sent over the network. This suggests the major bottleneck is not due to the I/O or the network exchange of a large number of chunks but because of the unoptimized array management.

We selected shape B for the aggregation, reshaping, wind speed, and window hyperslabbing queries. We benchmarked the *min*, *max*, and *avg* performance but show the results only for the latter due to the similarity of numbers and space constraints, fig. 4d. The result is the single 94 × 192 grid, section 4.3. ChronosDB outperforms SciDB from 3× to 5× for cold queries and from 6× to 10× for hot queries.

The reshaping $(time, lat, lon) \mapsto (lon, lat, time)$ was evaluated since the respective SciDB operator has a limitation: it does not accept the order of dimensions and completely reverses their order. The ratio is up to 26× (figs. 4g and 4h).

Wind speed (ws) at each grid cell and time point is calculated as $ws = \sqrt{\text{u-wind}^2 + \text{v-wind}^2}$. Climate models often do not produce ws and it is derived once required. The input and the resulting arrays have the same chunk shape leading to no actual shuffling. The ratio is up to 25× (fig. 4l). ChronosDB is 3× faster for the hot run on 1 node with RAMD (30 vs. 94 sec.): RAMD helps to avoid hitting SSD I/O limits.

The ratio slightly drops on 32 nodes: each ChronosDB worker has only 1 subarray and can use only 1 CPU core to process it; SciDB has ≈875 chunks per node in this case.

To contrast the performance resulting from an inappropriate chunk shape, we evaluate the hyperslabbing of the 20 × 20 window on both shapes A and B, figs. 4e and 4f.

Time series hyperslabbing (all time steps for the single point $(0, 0)$) was evaluated on shape D (fig. 4j).

ChronosDB hyperslabs the chunked arrays up to 3× faster than the original. This proves that inexact chunking is an I/O accelerator when the subarrays can be balanced between cluster nodes. However, ChronosDB is very fast in both hyperslabbing types even on the original array, fig. 4e. This is a strong advantage of ChronosDB.

We measured the performance of altering the chunk shape A ↦ B, A ↦ D, and B ↦ D, figs. 4a to 4c. The first two are to estimate the speed of switching from the initial shape to one of the most appropriate for the given workloads. The last is to estimate the speed of switching between the workloads. We used the fastest SciDB query to change the shape [71]. ChronosDB outperforms SciDB by up to 1034×.

Although SciDB and ChronosDB chunking are not precisely equivalent, they pursue the same goal: adapt to an

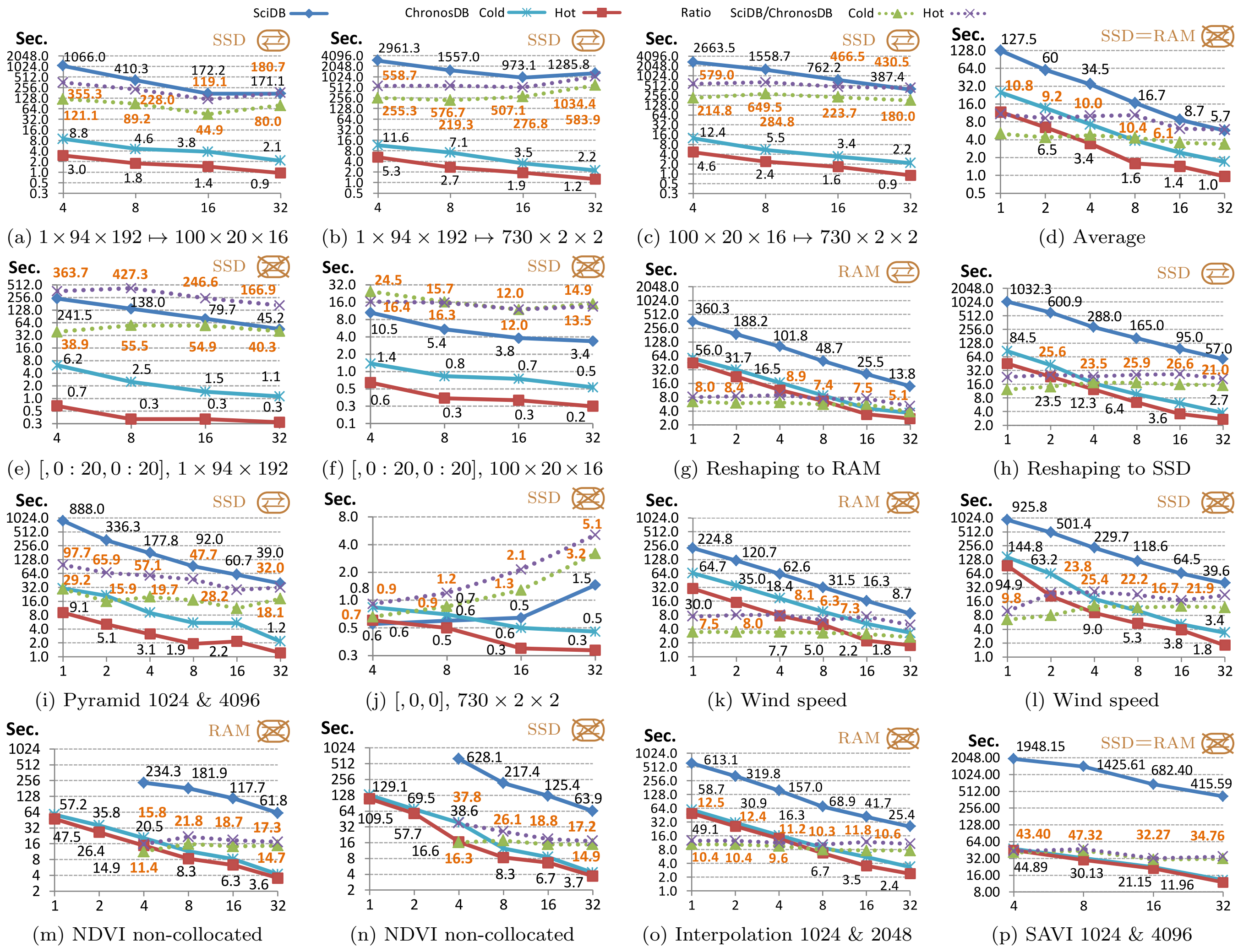


Figure 4: (a, b, c) chunking, (e, f) hyperslabbing $[0 : 46751, 0 : 20, 0 : 20]$, (j) hyperslabbing $[0 : 46751, 0 : 0, 0 : 0]$, a & b mean that $a \times a$ SciDB chunks and $b \times b$ ChronosDB subarrays were used. The horizontal axes plot the number of cluster nodes.

I/O workload (e.g. hyperslabbing). This dramatic runtime difference is probably not due to the network or disk I/O similarly to the aggregation as noted above and concluded from table 1 but due to the chunking algorithms themselves.

Fast NCO chunking performance is the result of years of hard work: NCO v4.4.2 (2014/02/17) chunks the original array to 730 × 2 × 2 about 100× slower than NCO v4.6.7. Certainly SciDB will improve its chunking technique but this will require time, development efforts, financial support, and other relevant investments. This is additional evidence to leverage already existing and elaborate command line tools.

ChronosDB keeps a large number of chunks in a much smaller number of subarrays (files). This multilevel cell storage approach makes it possible to perform chunking and adapt to workload much faster when the subarrays are balanced between cluster nodes. This is straightforward to achieve when the number of subarrays is equal to or a multiple of cluster nodes: the usual case in practice.

Unlike SciDB, ChronosDB is able to quickly adapt to changing workloads. It is also very fast for a ChronosDB user to try different chunk shapes for a given workload.

5.3 Landsat Satellite Data

Please, refer to section 3.1 for the dataset description.

We benchmarked each operation on arrays with SciDB chunk and ChronosDB subarray shapes 1024 × 1024, 2048 × 2048, and 4096 × 4096. We pick the shape on which a system performs best and show the respective runtime in fig. 4.

We benchmarked the 2× interpolation on the NIR array, fig. 4o. ChronosDB outperforms SciDB from 7× to 12×.

Note that ChronosDB is not only faster but it performs the real nearest-neighbor interpolation. In contrast, SciDB just fills new cells with the values taken from the cells with the coordinates known in advance.

We benchmarked the creation of 3 levels of the multiresolution pyramid also on the NIR array, fig. 4i. ChronosDB outperforms SciDB from 11× to 97×.

The calculation of NDVI demonstrates the K-way array join, section 4.2. We evaluated NDVI computation on RED array with shape 2048 × 2048 and NIR array with all three shapes. SciDB fails to compute NDVI on 1- and 2-node clusters with a `not enough memory error`. SciDB RED and NIR arrays are collocated when their shapes coincide and no exchange of chunks over the network is required to compute NDVI. This case yields the fastest performance (not shown).

Other pairs of shapes lead to partially collocated SciDB arrays. Figures 4m and 4n show the performance when ChronosDB subarrays are totally non collocated: DefaultPolicy (section 4.1) assigned NIR subarrays to workers in reverse order. This requires ChronosDB to send over the network all subarrays of RED or NIR array ($\approx$1.6–1.8 GB in total depending on the retiling parameters). ChronosDB triggers the retiling to calculate NDVI when the subarray shapes of RED and NIR arrays do not coincide, section 4.2.

ChronosDB is exceptionally superior (from $11\times$ to $21\times$) to SciDB even when SciDB discards its output, fig. 4m. SciDB and ChronosDB performed best on the 1024×1024 and 4096×4096 chunk and subarray shapes of the NIR array respectively. ChronosDB materialized all its intermediate data during the join operation to RAMD.

SciDB always fails on DS2 V2 VMs with a `not enough memory error` when computing the SAVI pipeline (section 3.4). We selected DS3 V2 VMs (14 GB RAM, 4 CPU cores). SciDB also fails on 1- and 2-node DS3 V2 clusters. SciDB and ChronosDB performed best with the same shapes as for the NDVI case. SciDB arrays were again partially collocated while ChronosDB arrays were totally non collocated. ChronosDB materialized the intermediate data to RAMD. It was about 30% slower without RAMD. ChronosDB is from $32\times$ to $47\times$ faster than SciDB, fig. 4p. This proves ChronosDB to be efficient for complex analytic pipelines.

6. RELATED WORK

Unlike all similar systems, ChronosDB combines a unique set of features. It (1) operates in situ on raster files, (2) runs on a computer cluster, (3) scales existing industry-standard tools, provides (3) a formal data model, (4) a clear command line query syntax, (5) an efficient execution engine, and (6) a rich set of operations (some are not listed due to space constraints: other NCO/GDAL tools and ImageMagic [29]).

Open source array DBMS. SciDB [7, 74, 73, 16] requires complex and slow data import. Array dimensions are integer making some operations difficult or impossible on many real-world datasets: time series, Gaussian grids, etc. SciDB functionality constrained the benchmark in section 5.

PostGIS is a popular industrial geospatial engine [51]. It imports files into BLOBs or registers out-db files resulting in a set of separate tiles not treated logically as a single array. PostGIS is not distributed and is tailored to 2-d tiles [50].

RasDaMan Community requires data import [2], works on top of PostgreSQL on a single machine, and is based on array algebra [4]. It allows expressing a wide range of operations but it may be far not straightforward [54, 55, 53]. It is easier to use dedicated functions for each operation. Only ChronosDB and RasDaMan data models are formalized.

SciQL is an array simulation on top of MonetDB but it is not in situ enabled and not under active development [91].

TileDB has a new on-disk format with support for fast updates (it is not very relevant for georeferenced arrays). TileDB is not yet distributed or in situ enabled. It does not yet offer any array processing functions [79, 78, 48].

Commercial array DBMS/GIS. RasDaMan Enterprise is in situ enabled and distributed with largely the same functions as the open version. It adds support for other base DBMS and performance accelerators [3, 52]. Oracle Spatial is distributed but not in situ enabled and tailored to 2-d arrays [46]. Its functionality is similar to PostGIS. ArcGIS ImageServer focuses on complex geospatial analytics as well as powerful visualization and publishing capabilities [1].

In situ algorithms. Blanas et al. proposed in-memory techniques [6], ArrayUDF scales out user-defined sliding window functions [17], and SAGA runs aggregation queries [86] over HDF5 files. FastQuery bitmap indexes can be stored alongside the original data [10]. DIRAQ reorganizes data for efficient range queries [32]. Su et al. proposed user-defined subsetting and aggregation over NetCDF files [75]. SciMate is optimized for several hyperslabbing patterns [85]. OLA-RAW performs parallel on-line aggregation of FITS files [9].

ArrayStore is not in situ enabled, focused on finding the optimal tiling strategy, and finally arrived at a similar regular tiling scheme to ChronosDB [72]. Google Earth Engine functionality covers the most of ChronosDB [25]. It does not operate in situ. Its native data model is a collection of 2-d images. It supports N-d arrays but less efficiently [24].

SWAMP is the most prominent effort to scale NCO. It partitions a bash script calling NCO tools and executes its parts on a computer cluster [82, 83]. Each cluster node usually stores a copy of all data. No data model is provided: a user explicitly iterates over a set of files and deals with file partitioning. ChronosDB abstracts from file names, their quantity, location on cluster nodes, and other details.

Ad-hoc software. Practitioners mainly develop batch-style scripts launched at each cluster node without data exchange or HPC programs to process data on a supercomputer via e.g. MPI [49]. ChronosDB is a good complement to the world of isolated scripts and MPI programs to streamline certain tasks of big geospatial data processing in the Cloud. Note that ChronosDB can scale out scripts as well.

National initiatives include Australian Data Cube [36], Russian UniSat [30], European EarthServer [5]. They feature complex analytic pipelines implemented using a broad range of software. ChronosDB or a similar system may serve as a building block, not an all-in-one substitute for these initiatives. In situ processing with popular command line tools suits best for interoperability with these ecosystems.

Hadoop-based approaches need data import. The plugin called SciHadoop has a driver for accessing HDFS-based arrays via NetCDF API [8]. Hadoop [27] and SciDB [70] streaming can feed data into a tool's `stdin` and ingest its `stdout`. Note three time-consuming data conversion steps: data import into an internal format, export into `stdin`, and import from `stdout`. ChronosDB delegation takes place without any data conversion. SparkArray extends Spark with filter, hyperslabbing, smooth and join array operations. It did not outperform SciDB for any query [84].

7. CONCLUSIONS

ChronosDB delegates portions of raster data processing to feature-rich and highly optimized command line tools. This makes ChronosDB run much faster than SciDB. In addition, ChronosDB design is highly abstracted and independent from underlying raster file formats and tools. Standard file formats are interfaces between ChronosDB and tools written in diverse programming languages and generally supported by active communities. The command line query syntax of ChronosDB is clear, easy to use, and already well-known by most users. Future work includes developing a Web GUI or a remote API for ChronosDB to make it publicly available as a Cloud service at chronosdb.gis.land.

8. REFERENCES

[1] ArcGIS for server — image extension. http://www.esri.com/software/arcgis/arcgisserver/extensions/image-extension.

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. Spatio-temporal retrieval with RasDaMan. In *VLDB*, pages 746–749, 1999.

[3] P. Baumann, A. Dumitru, and V. Merticariu. The array database that is not a database: File based array query answering in RasDaMan. In *SSTD 2013*, volume 8098, pages 478–483. LNCS, Springer.

[4] P. Baumann and S. Holsten. A comparative analysis of array models for databases. *Int. J. Database Theory Appl.*, 5(1):89–120, 2012.

[5] P. Baumann, P. Mazzetti, J. Ungar, R. Barbera, D. Barboni, A. Beccati, L. Bigagli, E. Boldrini, R. Bruno, et al. Big data analytics for Earth sciences: the EarthServer approach. *International Journal of Digital Earth*, 9(1):3–29, 2016.

[6] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, pages 385–396. ACM, 2014.

[7] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[8] J. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based query processing in Hadoop. In *SC 2011*.

[9] Y. Cheng, W. Zhao, and F. Rusu. Bi-level online aggregation on raw data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM, 2017.

[10] J. Chou, K. Wu, et al. FastQuery: a general indexing and querying system for scientific data. In *International Conference on Scientific and Statistical Database Management*, pages 573–574. Springer, 2011.

[11] Climate Wikience and ChronosServer. http://www.wikience.org.

[12] Climate Wikience and ChronosServer (mirror). http://wikience.gis.land.

[13] Conda package manager. https://conda.io/docs/user-guide/install/download.html.

[14] Coverity scan: GDAL. https://scan.coverity.com/projects/gdal.

[15] Coordinate Reference Systems (CRS) – Quantum GIS (QGIS) documentation. https://docs.qgis.org/2.14/en/docs/gentle_gis_introduction/coordinate_reference_systems.html.

[16] 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. *PVLDB*, 2(2):1534–1537, 2009.

[17] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu. ArrayUDF: User-defined scientific data analysis on arrays. In *HPDC*, 2017.

[18] ERDDAP – working with the datasets.xml file. https://coastwatch.pfeg.noaa.gov/erddap/download/setupDatasetsXml.html.

[19] GDAL homepage. http://www.gdal.org/.

[20] GDAL: "gdalinfo" tool. http://www.gdal.org/gdalinfo.html.

[21] GDAL: "gdalbuildvrt" tool. http://www.gdal.org/gdalbuildvrt.html.

[22] GDAL virtual file format. http://www.gdal.org/gdal_vrttut.html.

[23] GeoTIFF. http://trac.osgeo.org/geotiff/.

[24] Array overview: Google Earth Engine API. https://developers.google.com/earth-engine/arrays_intro.

[25] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore. Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 2017.

[26] M. Grawinkel et al. Analysis of the ECMWF storage landscape. In *13th USENIX Conf. on File and Storage Technologies*, page 83, 2015.

[27] Hadoop streaming. wiki.apache.org/hadoop/HadoopStreaming.

[28] A. R. Huete. A soil-adjusted vegetation index (SAVI). *Remote sensing of environment*, 25(3):295–309, 1988.

[29] ImageMagick. http://imagemagick.org.

[30] A. Kashnitskii, E. Lupyan, I. Balashov, and A. Konstantinova. Technology for designing tools for the process and analysis of data from very large scale distributed satellite archives. *Atmospheric and Oceanic Optics*, 30(1):84–88, 2017.

[31] Kernel.org tmpfs. https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt.

[32] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova. Scalable in situ scientific data encoding for analytical query processing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 1–12. ACM, 2013.

[33] Landsat 8 bands. https://landsat.usgs.gov/what-are-band-designations-landsat-satellites.

[34] Landsat naming conventions. https://landsat.usgs.gov/what-are-naming-conventions-landsat-scene-identifiers.

[35] Launching digitalglobe's maps API. https://www.mapbox.com/blog/digitalglobe-maps-api/.

[36] A. Lewis, S. Oliver, L. Lymburner, B. Evans, L. Wyborn, N. Mueller, G. Raevksi, J. Hooke, R. Woodcock, J. Sixsmith, et al. The Australian geoscience data cube—foundations and lessons learned. *Remote Sensing of Environment*, 2017.

[37] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *ACM SIGMOD Record*, volume 25, pages 228–239. ACM, 1996.

[38] A. P. Marathe and K. Salem. Query processing techniques for arrays. *The International Journal on Very Large Data Bases*, 11(1):68–91, 2002.

[39] Measuring vegetation (NDVI & EVI): Feature articles. https://earthobservatory.nasa.gov/Features/MeasuringVegetation/.

[40] S. Nativi, J. Caron, B. Domenico, and L. Bigagli. Unidata's common data model mapping to the ISO 19123 data model. *Earth Sci. Inform.*, 1:59–78, 2008.

[41] NCEP-DOE AMIP-II Reanalysis. http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis2.html.

[42] The NetCDF markup language (NcML). https://www.unidata.ucar.edu/software/thredds/current/netcdf-java/ncml/.

[43] NCO homepage. http://nco.sourceforge.net/.

[44] OGC (Open Geospatial Consortium) Network Common Data Form (NetCDF). http://www.opengeospatial.org/standards/netcdf.

[45] Oracle database online documentation 12c release 1 (12.1), Spatial and Graph GeoRaster developer's guide. https://docs.oracle.com/database/121/GEORS/geor_image_proc.htm.

[46] Oracle Spatial and Graph. http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html.

[47] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84, 2013.

[48] S. Papadopoulos et al. The TileDB array data storage manager. *PVLDB*, 10(4):349–360, 2016.

[49] Personal communications with the staff of German Climate Computing Centre (DKRZ), Space Research Institute (Russia), Institute of Numerical Mathematics (Russia), University of Madrid (Spain), Moscow State University, and other institutions.

[50] PostGIS: Handling N-d arrays. https://lists.osgeo.org/pipermail/postgis-users/2017-October/042433.html.

[51] PostGIS raster data management. http://postgis.net/docs/manual-2.2/using_raster_dataman.html.

[52] RasDaMan features. http://www.rasdaman.org/wiki/Features.

[53] RasDaMan forum: Condense operation. https://groups.google.com/forum/#!topic/rasdaman-users/28WWbNdTWYg.

[54] RasDaMan forum: Query on multiple collections. https://groups.google.com/forum/#!topic/rasdaman-users/Vu0V4Ed6zms.

[55] RasDaMan forum: Sobel filter. https://groups.google.com/forum/#!topic/rasdaman-users/fdu5jzQ9kmw.

[56] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[57] J. A. Richards. *Remote Sensing Digital Image Analysis: An Introduction*. Springer-Verlag Berlin Heidelberg, 5th edition, 2013.

[58] R. A. Rodriges Zalipynis. ChronosServer: real-time access to "native" multi-terabyte retrospective data warehouse by thousands of concurrent clients. *Inf., Cyb. and Comp. Eng.*, 14(188):151–161, 2011.

[59] R. A. Rodriges Zalipynis. ChronosServer: Fast in situ processing of large multidimensional arrays with command line tools. In *Supercomputing: Second Russian Supercomputing Days, RuSCDays 2016, Moscow, Russia, September 26–27, 2016, Revised Selected Papers*, volume 687 of *Communications in Computer and Information Science*, pages 27–40, Cham, 2016. Springer International Publishing.

[60] R. A. Rodriges Zalipynis. Array DBMS in environmental science: Satellite sea surface height data in the Cloud. In *9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2017, Bucharest, Romania, September 21-23, 2017*, pages 1062–1065. IEEE, 2017.

[61] R. A. Rodriges Zalipynis. Distributed in situ processing of big raster data in the Cloud. In *Perspectives of System Informatics – 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers*, volume 10742 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2018.

[62] R. A. Rodriges Zalipynis et al. Array DBMS and satellite imagery: Towards big raster data in the Cloud. In *Analysis of Images, Social Networks and Texts – 6th International Conference, AIST 2017, Moscow, Russia, July 27-29, 2017, Revised Selected Papers*, volume 10716 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 2018.

[63] SciDB consume() and chunk shuffling. http://forum.paradigm4.com/t/consume-and-chunk-shuffling/2056.

[64] SciDB output chunk distribution. http://forum.paradigm4.com/t/does-store-redistributes-chunks-among-cluster-nodes/1919.

[65] SciDB forum: Interpolation. http://forum.paradigm4.com/t/interpolation/1283.

[66] SciDB configuration. https://paradigm4.atlassian.net/wiki/display/ESD169/Configuring+SciDB.

[67] SciDB documentation: Join operator. https://paradigm4.atlassian.net/wiki/spaces/ESD169/pages/50856234/join.

[68] SciDB hardware guidelines. https://www.paradigm4.com/resources/hardware-guidelines/.

[69] SciDB hardware guidelines (archived copy). http://wikience.org/archive/SciDB_HW_guidelines_24Feb2018.png.

[70] SciDB streaming. https://github.com/Paradigm4/streaming.

[71] Scidb forum: The fastest way to alter chunk shape. http://forum.paradigm4.com/t/fastest-way-to-alter-chunk-size/.

[72] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: a storage manager for complex parallel array processing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 253–264. ACM, 2011.

[73] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[74] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013.

[75] Y. Su and G. Agrawal. Supporting user-defined subsetting and aggregation over parallel NetCDF datasets. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 212–219. IEEE, 2012.

[76] THREDDS – dataset inventory catalog specification. https://www.unidata.ucar.edu/software/thredds/v4.5/tds/catalog/InvCatalogSpec.html.

[77] L. Tianhua, Z. Hongfeng, C. Guiran, and Z. Chuansheng. The design and implementation of zero-copy for Linux. In *Eighth International Conference on Intelligent Systems Design and Applications, ISDA'08*, pages 121–126. IEEE, 2008.

[78] TileDB. http://istc-bigdata.org/tiledb/index.html.

[79] TileDB: Documentation. https://docs.tiledb.io/docs/.

[80] D. C. Tomlin. *Geographic Information Systems and Cartographic Modeling*. Prentice-Hall, 1990.

[81] A. van Ballegooij. RAM: A multidimensional array DBMS. In *EDBT Workshops*, volume 3268, pages 154–165. Springer, 2004.

[82] D. L. Wang, C. S. Zender, and S. F. Jenks. Efficient clustered server-side data analysis workflows using SWAMP. *Earth Sci Inform*, 2(3):141–155, 2009.

[83] L. Wang et al. Clustered workflow execution of retargeted data analysis scripts. In *CCGRID 2008*.

[84] W. Wang, T. Liu, D. Tang, H. Liu, W. Li, and R. Lee. SparkArray: An array-based scientific data management system built on Apache Spark. In *IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE, 2016.

[85] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A novel MapReduce-like framework for multiple scientific data formats. In *CCGRID*, pages 443—450, 2012.

[86] Y. Wang, A. Nandi, and G. Agrawal. SAGA: Array storage as a DB with support for structural aggregations. In *SSDBM 2014*.

[87] What is map algebra? – ArcGIS help. http://desktop.arcgis.com/en/arcmap/latest/extensions/spatial-analyst/map-algebra/what-is-map-algebra.htm.

[88] C. S. Zender. Analysis of self-describing gridded geoscience data with netCDF operators (NCO). *Environmental Modelling & Software*, 23(10):1338–1342, 2008.

[89] C. S. Zender and H. Mangalam. Scaling properties of common statistical operators for gridded datasets. *The International Journal of High Performance Computing Applications*, 21(4):485–498, 2007.

[90] C. S. Zender and D. L. Wang. High performance distributed data reduction and analysis with the netCDF operators (NCO). In *87th AMS Annual Meeting*, 2007.

[91] Y. Zhang et al. SciQL: Bridging the gap between science and relational DBMS. In *IDEAS*, 2011.