Philipps Universität Marburg

Fachbereich Mathematik und Informatik

Bachelorarbeit Informatik

Iterator-Based Processing of Raster Time Series

Verfasst von:
Sören Hoffstedt
Matrikelnummer: 2476088
Kontakt: soeren.hoffstedt@posteo.de

Betreuer:
Prof. Dr. Seeger
Arbeitsgruppe Datenbanksysteme

Abgabedatum: 22.02.2019

Selbständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Ort, Datum	Sören Hoffstedt

Deutsche Zusammenfassung

Rasterdaten sind von großer Bedeutung in Gebieten wie der Meteorologie, der Biodiveristäts- und Klimawandelforschung. Sie beschreiben ein Phänomen wie Geländehöhe, Oberflächentemperatur oder Bewölkung für einen Raum. Sie unterteilen den Raum in ein mehrdimensionales Gitter von Zellen, die jeweils einen Wert beinhalten, der das Phänomen beschreibt. Viele Phänomene sind besonders in ihrer zeitlichen Veränderung interessant. Mehrere Raster, die das gleiche Phänomen zu unterschiedlichen Zeitintervallen beschreiben, bilden eine Rasterzeitreihe.

Visualization, Analysis, and Transformation (VAT) ist ein Tool zur interaktiven Untersuchung von Biodiversitätsdaten und erlaubt die kombinierte Verarbeitung von Raster- und Vektordaten. Zwar sind alle Daten in VAT Zeitreihen, doch die Verarbeitung von Rasterdaten als Zeitreihe ist nur eingeschränkt möglich. Das liegt daran, dass VAT nur Anfragen zu Zeitpunkten unterstützt, da im Frontend nur ein Raster gleichzeitig angezeigt wird.

Diese Bachelorarbeit stellt ein Konzept zur *Iterator basierten Verarbeitung von Rasterzeitreihen* vor, kurz IPRTS (vom Englischen *Iterator-based Processing of Raster Time Series*). Es soll in VAT integriert werden und wurde für die Bachelorarbeit als Prototyp implementiert. Das Konzept teilt Raster in mehrere *Kacheln* auf, um sie unabhängig ihrer Größe verarbeiten zu können. Da Rasterzeitreihen aus einer Vielzahl an Raster bestehen können, setzt IPRTS auf einen *Iterator basierten* Ansatz. Operatoren verarbeiten Rasterzeitreihen, in dem sie über die Kacheln der Zeitreihe iterieren und sie somit nacheinander verarbeiten. Um Rasterdaten nicht unnötig zu laden, setzt IPRTS außerdem auf *Deskriptoren*, mit denen *lazy loading* umgesetzt wird. Deskriptoren beschreiben eine Rasterkachel durch ihre Metadaten, wie räumliche und zeitliche Informationen, Auflösung, einen Kachelindex und die Anzahl der Kacheln des Gesamtrasters. Außerdem stellen sie eine Funktion zur Verfügung, die die eigentlichen Kacheldaten lädt.

Anfragen an IPRTS bilden einen Operatorbaum, der bestimmt, wie Rasterzeitreihen verarbeitet werden. IPRTS unterscheidet zwischen drei Operatortypen: Quelloperatoren, verarbeitende Operatoren und verbrauchende Operatoren. Die Quell- und verarbeitenden Operatoren des Operatorbaums bilden einen modularen Iterator über Kacheldeskriptoren. Der verbrauchende Operator iteriert über diese Deskriptoren und verarbeitet sie abschließend.

IPRTS implementiert mehrere Rasterzeitreihen Operatoren. Der *Expression*-Operator erlaubt lokale arithmetische Operationen auf jeder Zelle eines Rasters. Der *Convolution*-Operator berechnet den Wert einer Zelle aufgrund mehrerer Nachbarzellen, z.B. um Kantendetektion auszuführen, und setzt somit fokale Rasteroperationen um. Der *Raster-Value-Extraction*-Operator kombiniert Raster- mit Vektordaten. Der *Temporal-Aggregation*-Operator fasst mehrere Raster in Zeitintervallen zusammen und berechnet z.B. das Maximum jeder Zelle. Der *Temporal-Overlap*-Operator erzeugt für jede Überschneidung von Rastern aus zwei Zeitreihen ein Ausgaberaster.

Tests, die mit dem implementierten Prototyp von IPRTS durchgeführt wurden, zeigen, dass das Konzept funktioniert und gute Performance in der Verarbeitung von Rasterzeitreihen bietet.

Contents

1 Introduction

Raster data is widely used to represent geographical data in fields like meteorology, biodiversity, or climate change research. It describes phenomena in space and can measure, for example, temperature, precipitation, height, and more [1]. Rasters are a multi-dimensional grid of cells, and each cell contains a measurement of the phenomenon.
A raster is valid in a time interval. Multiple rasters describing the same phenomenon for different time intervals form a *raster time series*. Many phenomena described by rasters, like temperature or cloud coverage, are non-static and specifically, their change over time is of research interest. Thus, not only single rasters but raster time series must be analyzed.
Raster time series are not only of interest as they are but require processing for research and presentation purposes. The data can be analyzed statistically or combined with other data. For example, on single rasters, edge detection algorithms can be applied to analyze the distribution of a phenomenon. An example of processing multiple time series describing different phenomena is finding suitable locations to build solar parks. To thoroughly analyze the suitability of locations, different phenomena like solar radiation, cloud fractions, and terrain information can be analyzed using raster data. The distribution of clouds and solar radiation are important over time. Analyzing a single raster does not capture the suitability of a place sufficiently because it describes the phenomenon only statically. The mentioned raster time series can, for example, be aggregated to calculate the average sunshine duration or cloud coverage in a year. In the end, these results can be combined for a selection of the best locations for solar parks.
A single raster captured with modern satellite technology can be multiple gigabytes in size and raster time series can contain hundreds or thousands of rasters. To allow processing of rasters independent of their size, they can be divided into multiple smaller tiles.

The Visualization, Analysis, and Transformation system (VAT)[1] is a tool for interactive exploration of biodiversity data [2]. VAT is developed by the Database Research Group of the University of Marburg [2] and is part of the GFBio project[3], a German data infrastructure for biological and environmental research. VAT supports both raster and vector data in the same environment. It can process them individually and combined. The web-based front end of VAT is called WAVE and provides intuitive workflows for processing and visually analyzing data. To make exploratory research reproducible, it keeps track of the users processing steps and citations of the used data. To provide fast processing and a responsive visualization, VAT processes the raster and vector data in the back end MAPPING.
Every raster and vector object in VAT has a temporal validity, and therefore each dataset is

[1] `https://vat.gfbio.org/`
[2] `https://www.uni-marburg.de/fb12/en/researchgroups/dbs`
[3] `https://www.gfbio.org/`

a time series. However, the possibilities for processing rasters as a time series are limited. Queries to MAPPING can not request data for a time interval but for a time point because only one raster can be displayed in the front end. Thus, the support for operations that process rasters as a complete time series and not as individual rasters from a time series is limited. VAT supports operations like expressions, plotting of histograms, or raster value extraction. It also has an operator for temporal aggregation of a raster time series. It processes a raster time series and not just individual rasters. Its implementation is based on a workaround that adds a time duration to the parameters. The requested time point and the duration build a time interval and the operator aggregates all rasters of the time series that fall into it. To allow more complex operations on raster time series, the concept of processing raster data in MAPPING has to be extended to allow operations on time intervals.

This Bachelor thesis explores a concept that improves the processing of raster time series. The design and prototypical implementation of a concept called *Iterator-Based Processing of Raster Time Series (IPRTS)* will be introduced. It is developed with the goal to be integrated into MAPPING and is therefore designed to process queries in with very short response times. IPRTS is an *iterator-based* processing system that divides a raster time series into raster tiles and processes them individually in succession. This allows executing operations on raster time series independent of their size. IPRTS inherently processes raster time series as a time series and not as single rasters. Additionally, IPRTS uses lazy loading of raster data to support operations that change the temporal resolution of a raster time series. Therefore, it processes descriptors of raster tiles that contain *metadata* about the tile and a method to actually execute the loading of the tile's grid cells. This thesis focuses on exploring the concept of raster time series processing and does not include important and adjacent topics like performance optimized storage of raster time series.

The rest of this thesis is structured as follows. Section 2 defines the basic concepts about raster time series that are used throughout the thesis. Section 3 analyzes the problems that have to be solved by a raster time series processing system. Section 4 takes a look at different software systems and their handling of raster time series processing. Section 5 describes the concept and design of *IPRTS* and Section 6 discusses its implementation. Section 7 presents an example query to *IPRTS* and its result. Section 8 presents benchmarks that explore the performance of the implementation of *IPRTS*. Finally, Section 9 concludes the thesis.

2 Basics

2.1 Raster, Tile, and Raster Time Series

A *raster* is a multi-dimensional grid of *cells*. They describe a geographical phenomenon in space. All grid cells (also called pixels) of a raster contain a value of the same data type. They represent a specific unit that is associated with the captured phenomenon. Rasters are mostly two- or three-dimensional, but for simplification and presentation purposes, only two-dimensional rasters will be discussed in the following.
The *resolution* of a raster is the number of cells it contains in each dimension. The space covered by a raster is called *spatial domain* and is defined by a *coordinate reference system* (CRS), also called *projection* [3]. A projection maps locations on the globe onto a two-dimensional plane and uses a specific coordinate system for referencing locations. A spatial domain can be described by four coordinate values that represent the borders of the spatial rectangle. *Nodata* is a special value in the range of the raster's data type. A special value in the range of the rasters data type is the *nodata* value. Each cell that contains the nodata value does not have information about the phenomenon. Mathematically, a raster can be defined by a function that maps from a coordinate set to a value set [4].
The cells are a division of the spatial domain of the raster into discrete, equally sized units. Rasters describe in most cases continuous phenomena, but the division of space into cells makes their description discrete. Thus, rasters are an approximation of the phenomenon, and its quality is based on the raster's resolution and capturing technique. Additionally, raster cells describe not a single point in space but a rectangular space. Their size is derived from the resolution and spatial domain of the raster. Cells of a raster can be referenced in two ways: by their spatial domain in coordinate space and by a two-dimensional integer index that describes the cell's position in the grid of the raster.
Rasters can be divided into smaller grids of cells called *tiles*. Especially because high-resolution rasters are huge, processing raster tiles individually might be necessary because of memory limitations. Tiles of a raster do not overlap and can be of regular or irregular size and positioning. Conceptually, a tile is the same as a raster. Both describe a phenomenon in space with grid cells. The difference is that a tile is part of a raster and a raster can be divided into tiles.
For phenomena that change over time, it is of interest to capture this change. Therefore, rasters capturing the same phenomenon at different times form a *raster time series*. Generally, a time series is a collection of data points that are ordered by time. Every data point has a time interval that describes its validity. A start and an end time define a time interval. In a regular time series, the time interval between data points is always the same, while an irregular time series has varying time intervals. The starting time of the first data point and the end time

of the last data point form the temporal validity of the complete time series. If not stated otherwise, *time series* refers to a *raster time series* in this paper. The data points of a raster time series are rasters. The time interval a raster is valid in is called *temporal domain*. Even though raster time series can have irregular time intervals, they often have regular intervals, for example, for data captured by satellites.

This thesis works with two assumptions about time. First, a raster time series does not have gaps. This means that the end time of a raster is equal to the start time of the following raster. Therefore, the end points of a time interval are exclusive to the validity of a raster. Second, rasters of a time series do not overlap temporally. Referring to the *next* or *previous* raster is based on the temporally ascending order of the rasters.

Vector data is another format of representing geographical information. Vector formats use geometries like points, lines, or polygons to describe geographical entities in coordinate space [1]. These entities are described by attributes of different types, like a name, a time interval, or values of a measurement. An geometry instance with associated attributes is called a feature. Multiple features of the same geometry with the same attribute types can form a collection.

2.2 Raster Operations

To analyze raster data, operations can be applied to one or multiple rasters. An operation transforms the cells of an input raster and creates, as a result, an output raster. Simple examples are arithmetic expressions that multiply each cell value by 2 or add the cells of two rasters together.

Operations on raster data have been formalized by different authors, using names like raster, map, or array algebra [4] [5]. C. Dana Tomlin categorizes raster operations into three distinct types: local, focal, and zonal operations [3]. These are the operation types that are required

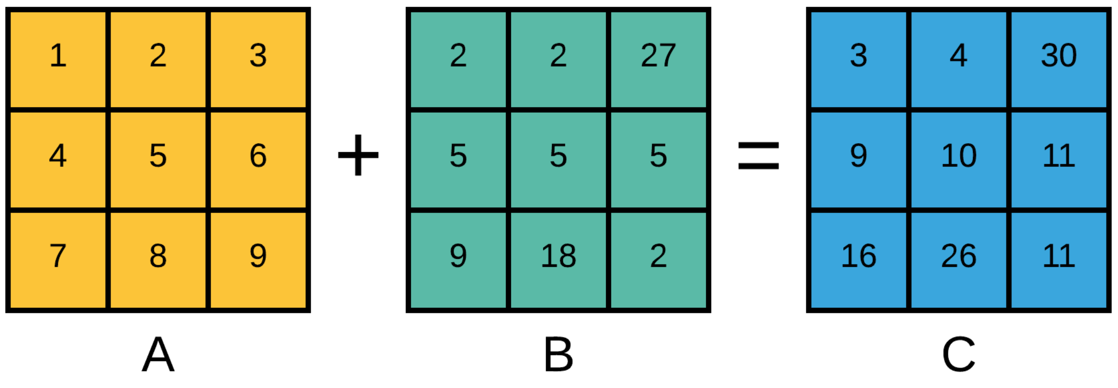


Figure 1: Arithmetic expression on cell level for 3×3 rasters: $A + B = C$

for a flexible raster time series processing system. If not stated otherwise, these operations take one or multiple rasters as operands and create a new raster as the result.

Local operations calculate a result for each cell only taking that cell as an input. The result is a new raster of the same spatial extent. They can be based on a single or multiple input rasters. Possible operations can be derived from the arithmetic operations of the rasters data type. An example of a local operation on a single raster is dividing the cells of a raster by a maximum value to normalize them. Figure 1 shows a simple example of adding two small rasters.

In the context of a raster time series, local operations can be executed differently. First, it can be an arithmetic operation between rasters from two different time series. The result is a raster time series where, for example, the rasters of one time series were subtracted from rasters of the other time series. Second, it can an operation aggregating multiple rasters of one time series together, for example, calculating the yearly mean from a monthly time series.

Focal operations do not calculate the value of an output cell solely based on the same input cell but also include its neighboring cells. Depending on the use case, each cell is differently weighted. The definition of the cell neighborhood that has to be used in a calculation and the weight of each cell are defined is called kernel or template. Figure 2 demonstrates how applying a 3×3 smoothing kernel on the central cell of an input raster calculates an output cell.

Focal operations are also called *convolution functions* [4] or *template operations* [4]. They are used for edge detection, raster smoothing, sharpening, or by convolutional neural networks in the field of deep learning.

Zonal operations divide a raster into zones and calculate values for each zone. A zone can be defined by a polygon or raster where each cell contains a zone identifier. They can represent, for example, borders of countries. The calculation is based on the cell value and existing

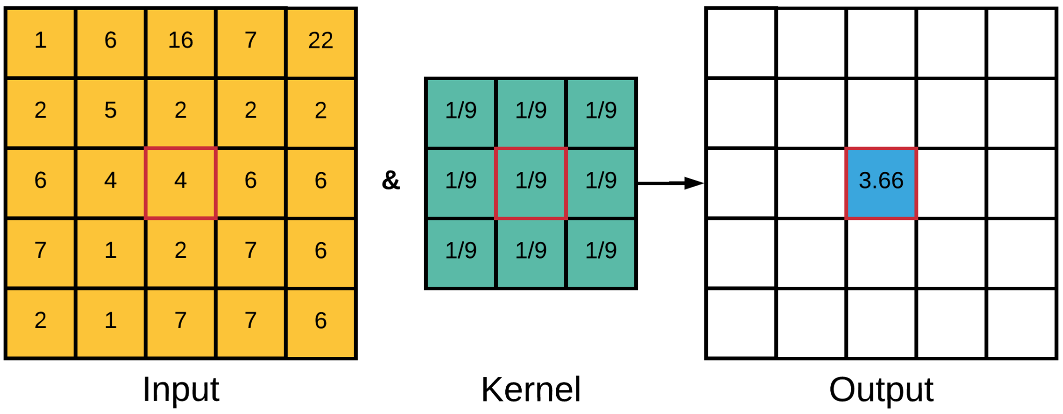


Figure 2: Focal operation: Applying the kernel on the central cell (red).

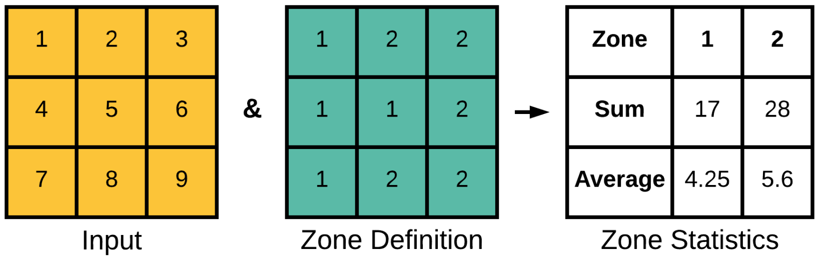


Figure 3: Zonal operation: Calculating statistics for zones.

values associated with its zone. Zonal operations can, for example, be used for calculating the maximum or average cell values for each zone. Figure 3 shows an example where indices in a second raster define the zones. The output calculates the sum and average of all cells in each zone.

2.3 Raster Time Series Operators

Raster time series operators process one or multiple time series in different ways. In the following, five operators will be introduced that each represents a core functionality for a processing system of raster time series: *expression, convolution, temporal aggregation, temporal overlap,* and *raster value extraction.* These operators include each raster operation type and operators that expand and condense the temporal resolution of a raster time series. They will be used as proof of concept operators for implementation of IPRTS.
A query is a request for raster data. In the context of a query based processing system for raster time series, a query rectangle describes the spatial-temporal domain of the time series that is the result of the query. A *query raster time series* and *query rasters* are the results of a query. A *source time series* and *source rasters* are the input rasters that are processed by a query.

2.3.1 Expression

The expression operator provides local arithmetic operations on one or multiple rasters with the same resolution. Operands can be rasters and numeric values, but one of them must be a raster. The result of an expression is a raster. The operator should support different operations like addition, subtraction, division, multiplication, or modulo. With raster being identified by capital letters, the operator must support expressions like *A + B*, or *A * 3*. As Figure 1 demonstrates, expressions are local operations where the result of each cell is only based on the same cell in the input rasters.
Expressions can be applied in three ways to raster time series. The first way calculates an

expression between rasters taken from different time series. The expression $A + B$ means that A and B represent rasters taken from the different time series. For example, the first, second, and so on rasters from both time series are taken as pairs for calculating the expression. The second and third way operate on one raster time series. If the expression only contains one raster operand, the expression will execute the calculation on every raster of the input time series. If the expression contains two raster operands, it can be executed on raster pairs taken from the one input time series.

2.3.2 Convolution

The convolution operator executes a focal operation on rasters. A kernel defines the way the value for each cell is calculated based on its neighborhood. Figure 2 shows a kernel that smooths the cells based on a 3×3 neighborhood. Other kernels can, for example, be used for edge detection. In the context of a raster time series, the operator executes the convolution on every individual raster.

2.3.3 Temporal Aggregation

The temporal aggregation operator aggregates rasters of a time series in time intervals. The aggregation can use different functions like mean, minimum, maximum, or sum. These are local raster operations executed over multiple input rasters from the same time series.

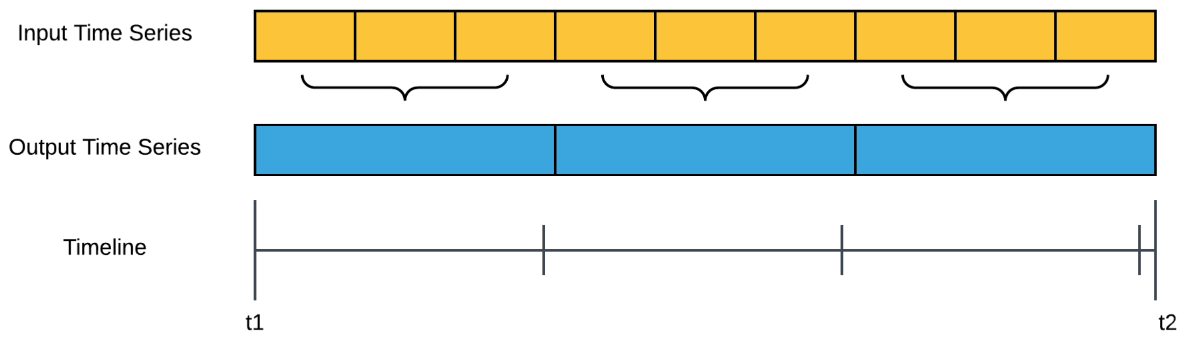


Figure 4: Temporally aggregating rasters of an input time series together by, for example, calculating the mean of three input rasters each.

Figure 4 demonstrates an example where the aggregation interval covers three rasters. Every three rasters will be aggregated into one output raster. In the figure, every colored rectangle is a raster and their width shows the temporal validity of the raster. Usually, the result will be a raster time series with fewer rasters than the input time series. When the time interval for aggregating is bigger than start and end time of the time series, all rasters of the time series will be aggregated together into a time series containing only one raster.

2.3.4 Temporal Overlap

The temporal validity of raster from two time series can be different. For example, raster time series A that as a temporal resolution of 15 minutes is combined with raster time series B that a resolution of 10 minutes. A combination of these time series requires the handling of their temporal overlap of both time series. Therefore, every overlap of two rasters from both time series must create an output raster. The temporal overlap operator does this as demonstrated in Figure 5. It temporally maps rasters from two input raster time series onto each other. As before, each colored rectangle is a raster from a time series and the width shows their temporal validity. Because of the temporal alignment, two input time series with a regular time interval of raster validity can create an irregular output time series as the figure shows. The calculation of the values of the output rasters is a local operation and must be definable like an expression.

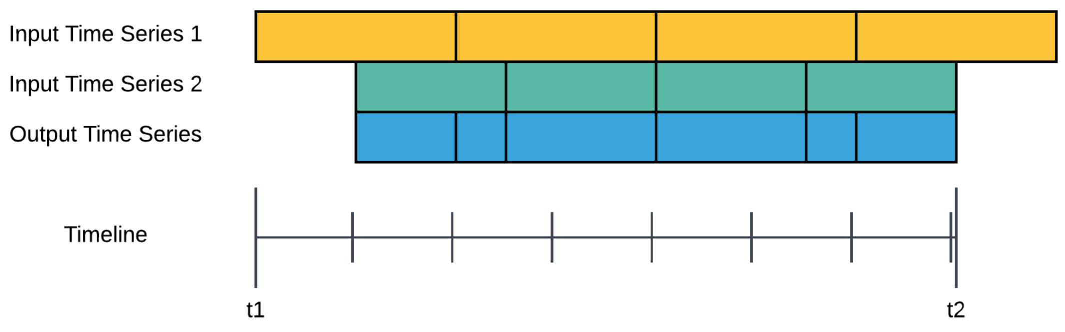


Figure 5: Temporal validity input and output time series for the temporal overlap operator.

2.3.5 Raster Value Extraction

The raster value extraction operator differs from the other operators by not outputting a raster time series. It is an interface between raster and vector data and demonstrates the strengths of VAT by combining different data types. The operator takes a list of spatial-temporal points as input and returns a cell value for each point by mapping them to a raster of an input time series by their temporal validity. If points are valid for a time interval, they will be mapped to a rasters time interval by their starting time. To map points to a cell of the raster, they must be in the coordinate space of the raster. When the operator is integrated with a system for processing vector data, it adds the cell values to the vector point collection as attributes. Extracting cell values for individual points can be used, for example, for moving objects to correlate the cell values with another attribute.

Raster value extraction is an important operator because it demonstrates a use case that does not output a raster time series. It combines raster and vector data. Even though raster value extraction is not directly a zonal operation, supporting it demonstrates that a processing system can handle transforming raster time series into different data formats. A zonal operator would have to map cells to zone information, while the raster value extraction maps cells to points.

3 Problem Analysis

Section 1 explained the importance of raster time series processing. With technological progress, the possibilities of capturing raster data have gotten better and faster. Data can be captured in higher resolution, both spatially and temporally, and the amount of available raster data continues to grow.
An example of a huge raster time series is provided by the *Meteosat Second Generation* satellites that capture meteorological raster data [4]. Satellite 11 is operated since 07/15/2015 and captures a raster every 15 minutes. In three and a half years of operation, it produced around 120000 rasters. A limited amount of the latest rasters can be downloaded publicly. These rasters are around 50 MB in size. As a time series covering three and half years of rasters would result in 6 terabytes of raster data.
But even single high-resolution rasters can be multiple GB in size. As an example, the Shuttle Radar Topography Mission (SRTM) provides the *1 Arc-Second Global* elevation data for most of the globe's landmasses [5]. In total, the data has a size of 25 GB and is publicly available as 1000 individual rasters that each cover parts of the globe.
These examples show that there is virtually no limit to the size of a raster time series. They can easily reach sizes that do not fit into the main memory of a computer and thus require sophisticated strategies for their storage and processing. Additionally, accessing data from non-volatile storage or network connections is the slowest IO step in data processing.
These observations form three requirements for a system that processes raster time series. 1. To support raster time series of any size, their rasters must be processed *iteratively.* 2. Due to the potentially huge size of single rasters, they must be divided into multiple *tiles.* 3. To avoid unnecessarily loading raster data from disk, the system has to utilize *lazy loading.* By providing *metadata* about rasters, operators must be able to decide if they need to access the data of a raster.
Raster value extraction is an example of an operator that benefits from these requirements. Depending on the spatial and temporal distribution of the points, single tiles or complete rasters of the time series might not actually be needed. The decision if pixel values must be read from a given raster tile can be made by looking at the spatial and temporal domain of its metadata. If no point overlaps with the tile, its data does not have to be accessed, and because tile data is loaded lazily, it was not read from disk unnecessarily.

[4] `https://www.eumetsat.int/website/home/Satellites/CurrentSatellites/Meteosat/index.html`
[5] `https://www.usgs.gov/centers/eros/science/usgs-eros-archive-digital-elevation-shuttle-radar-topography-mission-srtm-1-arc`

Other examples are use cases where the accuracy of processing results might not be as important as processing speed. This can be the case during experimental research on time series that have a very high temporal resolution. Instead of processing the time series completely, a sampling approach can reduce the number of processed rasters before the actual calculations. This allows for faster processing which is especially important in iterative and exploratory workflows as they are provided by VAT.

4 Existing Systems

While the suite of software for processing raster data is extensive, software with substantial support for the processing of raster time series is more limited. In Section 1, the VAT system was already introduced. It handles all data as time series, but the processing of raster time series is currently limited. Data in VAT can only be processed for a time point and not a time interval, and thus it only creates single rasters as output. The processing of raster time series requires to process all raster of the time series together and not individually. This limits the possibilities of processing the rasters as a time series. VAT supports many operations on raster time series like raster value extraction or expressions. It also supports temporal aggregation, but the implementation of the operator is based on a workaround to allow accessing multiple rasters of a time series.
In the following, three systems and their support for query-based processing of raster time series will be discussed: rasdaman, SECONDO, and GeoTrellis. All these systems do not support features that are important for VAT. For example, none of them support provenance tracking. As the workflow of VAT is built around iteratively adding new processing steps to data that is already visualized in the front end, caching processing results is an important feature of the back end MAPPING. It is also not inherently supported by the three systems.

4.1 Rasdaman

Rasdaman is a database management system specialized in storing and processing multi-dimensional arrays [6]. It is domain-independent but provides several features geared towards geographical raster data, especially by providing implementations of the Web Coverage Service (WCS) and Web Coverage Processing Service (WCPS) standards defined by the Open Geospatial Consortium (OGC)[6].
The concept for storing multi-dimensional array data is generalized as *datacubes*. Each dimension of a datacube can be defined by a CRS [7], allowing to define a three-dimensional array as a raster time series by setting the first two dimensions as a spatial reference system and the third

[6]`https://www.opengeospatial.org/`

as a temporal dimension. Each dimension can be defined either regular or irregular. Additional to importing raster data into traditional database storage, rasdaman allows for referencing external files, e.g., raster files in the GeoTiff format. Rasdaman supports storing rasters in regular and irregular tiles. Irregular tiling allows optimizing the data layout for specific access patterns.
The rasdaman server optimizes queries to provide efficient processing times [6]. Processing optimizations include query rewriting, splitting of queries into multiple sub-queries, distributed processing, and usage of multiple CPU cores and the GPU. Queries can be send to a rasdaman instance via various APIs. Java and C++ libraries allow for SQL-like queries and requests with WCS or WCPS are web-based. Operations supported by rasdaman are formalized by the *array algebra* [5], supporting all common local and focal operations, including aggregation and edge detection. For example, aggregation can be executed with the condenser operator that allows manipulation of the array dimensionality. It allows aggregation into single values like averaging every cell in a raster time series, but also aggregation into two-dimensional rasters.
As an array database, rasdaman does not support vector data.

4.2 SECONDO

SECONDO is a modular and extensible database management system [8]. By implementing algebra modules in C++ SECONDO can be extended to provide different data models and operations. It supports databases for moving object like trajectories, raster data, and more. Queries to the database can be defined in a SQL-like language (for relational databases) and execution language of the databases kernel. The SQL queries will be optimized while translated into the execution language. Using the execution language directly gives more detailed control about the actual operator execution. *Distributed SECONDO* is a distributed version of the database management system that utilizes the database *Apache Cassandra* to provide distributed data storage and query processing [9].
SECONDO provides algebra modules for working with raster data, mainly the modules *Raster2*, *Tile* [10], and *GIS* [11]. They allow loading raster data from different file formats like GeoTiff. Operations provided on raster data include temporal and spatial subsetting, retrieving meta information, and local operations on cells of a tile. Operations like temporal aggregation or edge detection are not directly supported, even though a query providing aggregation of rasters is definable from the other operators. The *GIS* module provides operators for terrain analysis, some of which are focal operations. For example, the *slope* operator calculates maximum slope a cell has to all its neighbors.
SECONDO does not provide support for geographical vector data.

4.3 GeoTrellis

GeoTrellis is a library written in Scala for processing raster data [12]. By utilizing the distributed cluster-computing framework *Apache Spark*, it allows processing large amounts of raster data.
GeoTrellis provides different back ends for storing raster data, like Amazon S3 cloud storage, Apache Cassandra, or the local file system. It supports all types of raster operations and implements convolution, many arithmetic operations, and more.
All raster objects are built on top of the *Resilient Distributed Dataset* data type of Apache Spark that handles the distributed storage and processing. The user can define the layout that stores the raster or raster time series in a distributed way. Single Rasters and a time series are both handled as the same basic data type but are based on different generic *key types* parameters. A key type determines the layout by, for example, being two- or three-dimensional. A raster is identified by a *SpatialKey* and a time series by a *SpaceTimeKey*. A corresponding key identifies specific subsets of a time series.
Rasters support *map, foreach, combine* operations that iterate over its tiles. These functions can be used to change the temporal layout of a time series, for example, by taking away the temporal component to aggregate a raster time series. Additionally, it supports exporting rasters, operations involving vector data, and handling of GeoTiff files by, for example, streaming raster data from them.

5 Concept

Iterator-Based Processing of Raster Time Series (IPRTS) is a concept for processing raster time series. It is designed to be integrated into MAPPING. Therefore the concept must be able to process queries with very short response times. A query defines an operator tree that can contain multiple operators for processing the time series in various ways. IPRTS realizes the three requirements for a raster time series processing system that were raised in Section 3. It divides rasters into *tiles* and processes them individually in operators. An operator *iterates* over all tiles of a raster time series to allow processing of huge raster time series. The loading of tile data is deferred until the data is really needed. To allow this, operators process *descriptors* that each contain metadata about a raster tile that they describe. On request, the tile data can be loaded from a descriptor. In the following, the core concept of IPRTS is presented in detail.

5.1 Operator Tree

Queries to IPRTS define an operator tree. Operators create *descriptors* based on input descriptors from any number of input operators. IPRTS has three types of operators: *source operators*, *processing operators*, and *consuming operators*.

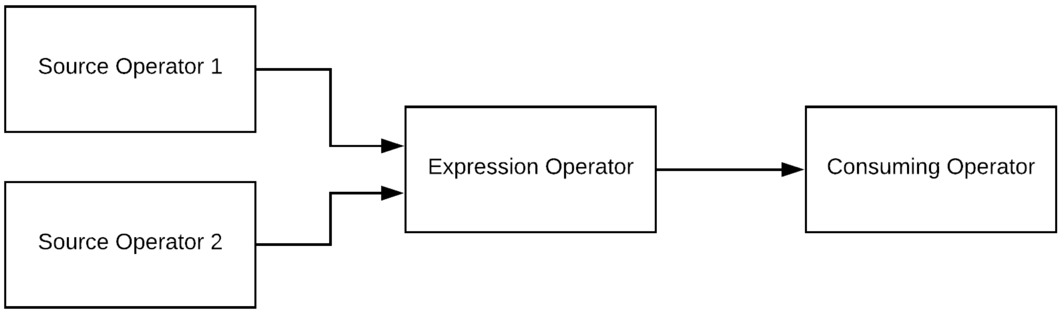


Figure 6: Example of an operator tree with two source operators.

The leaves of an operator tree are *source operators*. They create descriptors without input and load the raster data into IPRTS, for example, by loading raster files from disk. Source operators are the only operators that don't have input operators.
The root of an operator tree is a single *consuming operator*. It processes the tiles created by a single input operator but does not create output tiles. An example is an export operator that saves the time series to disk or a raster value extraction operator.

In between, any number of *processing operators* can be used for processing the rasters as needed. They have one or more input operators and create output descriptors based on their input descriptors. Figure 6 shows an example operator tree. It consists of two source operators, an expression operator that processes both time series, and a consuming operator.
Operators must be able to change parameters of their input operators, and each operator must be cloneable to allow query subsetting. An operator is cloned including all of its input operators, and the original and cloned version must be independent of each other. This way, the operator tree allows for query subsetting. An operator could, for example, divide its query into several sub-queries to execute them in parallel in separate operator instances.

5.2 Raster Tiles

To allow processing of very large rasters, IPRTS divides them into tiles. In each query, tiles are of a fixed and uniform size. As operators are iterating over tiles and not complete rasters, a fixed tile size and positioning allows operators to process tiles without checking the overlap and size differences between multiple tiles. This has some disadvantages, like not being able to optimize the tile size to take advantage of the tiling of different source rasters on disk. But every operator that handles operations between multiple raster time series would have to handle different sized and positioned tiles individually. Therefore, IPRTS uses tiles of a fixed size and position.
The fixed positions of tiles are achieved by dividing rasters from the same origin. The origin is based on the raster's projection. This means that the upper left tile is positioned to the origin of the projection and all tiles are offset from there in multiples of the tile size. Because source and query rasters do not have to cover the whole extent of the projection, not all possible tiles

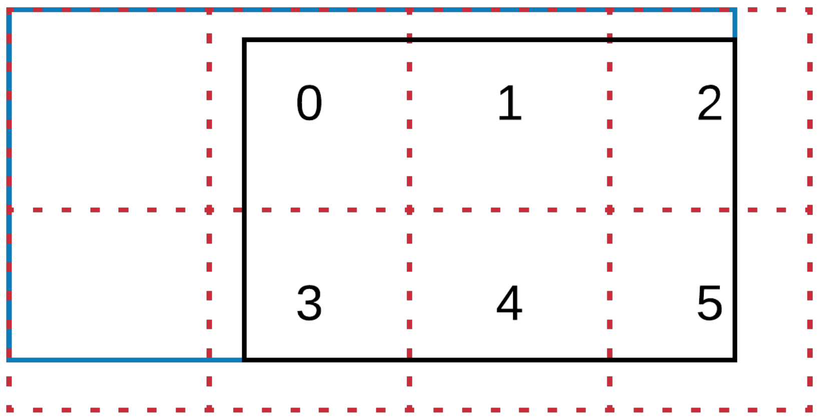


Figure 7: Tile indices of a query raster (black rectangle).

have to be processed but only the tiles that contain the requested spatial domain. Additionally, the tiles at the edges of the query raster can cover more space than the actual query rectangle if it does not align with the tile positioning. The additional data of the tiles either contains the regularly loaded raster data or is filled with nodata. In some cases, this design divides the query rectangle in more tiles than optimally necessary.

Most operators in the operator tree create a raster time series for the same spatial-temporal domain. This allows different raster time series to be used together without any overhead. An exception are operators that explicitly handle input rasters of a different spatial or temporal domain. When an operator processes incoming tiles from two different operators always in pairs, the tiles will always be for the same spatial position even when the source rasters of both have a different origin. When the source rasters of a query have different resolutions or projections, the difference has to be adjusted in the source operator. When an operator requires a different query rectangle between different input rasters, it can change the parameters of its input operators or create a sub-query. But it must make sure to return the rasters as expected by its input operators. Additionally, equally sized and positioned tiles allow caching tiles across multiple queries of different spatial domains.

The position of tiles in the processing order of a raster (not of the complete time series) is identified by the tile index. Index 0 describes the first tile of a raster that was processed. This can support different orders of processing the tiles of a raster, like z-order curve, Hilbert curve, or row-major order. Figure 7 show tiles of a single raster indexed in row-major order, how tiles

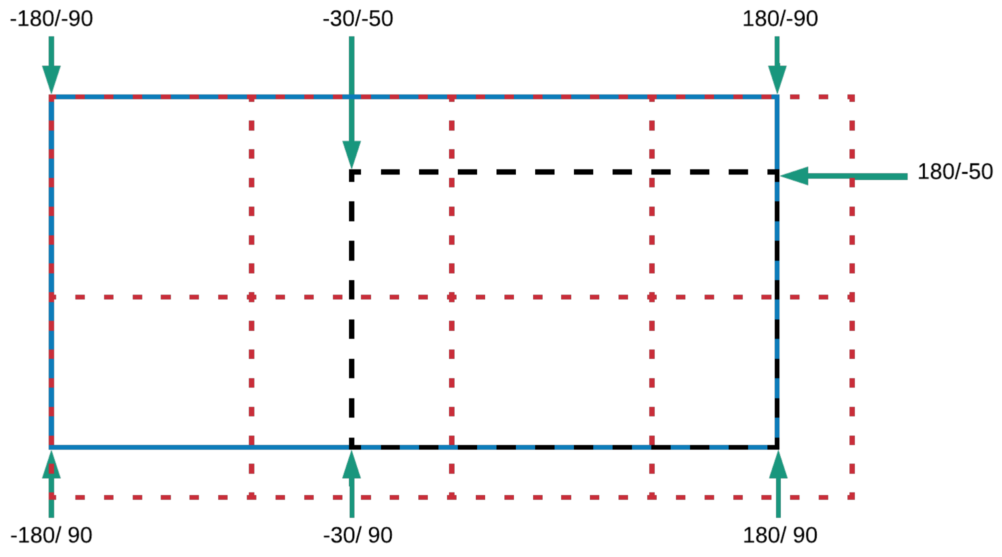


Figure 8: Blue: Extent of EPSG:4326 raster $(-180, -90, 180, 90)$, total pixel size of 3600×1800. Red dotted: Fixed tiles of size 1000×1000. Black: Query size of coordinates $(-30, -50, 180, 90)$, pixel size 2100×1400

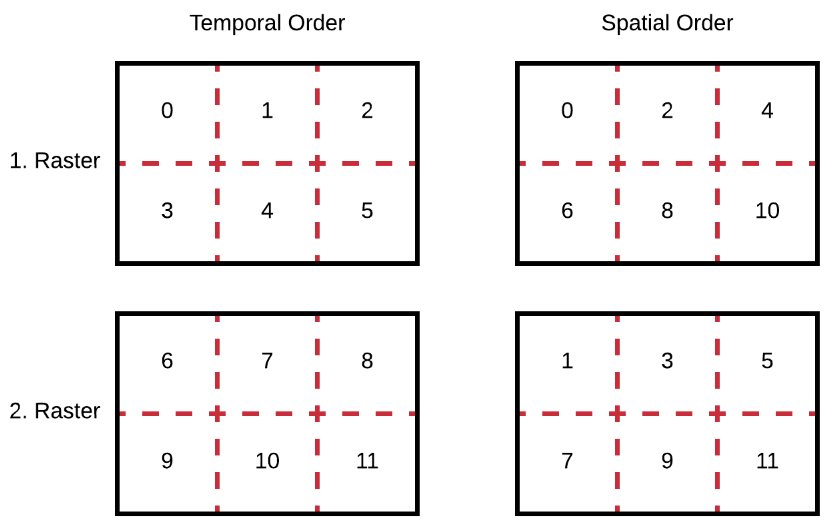


Figure 9: Temporal and spatial processing order of tiles from a two raster timer series.

outside of the query rectangle are not indexed, and tiles that are not fully covered by the query. The black rectangle shows the spatial domain of the query, the blue rectangle is the spatial extent of the raster's projection, and the red dotted rectangles are all possible tiles.

Figure 8 visualizes the positioning of tiles and covering more data than the query raster contains in more detail. The blue rectangle is a raster with EPSG:4326 projection and a resolution of 3600×1800 pixels. It covers the complete world with the coordinates $(-180, -90, 180, 90)$. The dotted red rectangles show all possible tiles, each 1000×1000 pixels in size. At the right and the bottom, it is visible that the edge tiles cover more pixel than the size of the source raster. The dotted black rectangle is a example query rectangle of the coordinates $(-30, -50, 180, 90)$. In the source raster, this rectangle has a size of 2100×1400 pixels. A source operator of IPRTS would create six tiles for this query, each also containing pixels that are not part of the actual query raster.

The most significant advantage of processing fixed sized and positioned tiles is that operators can process the tiles of a raster time series in two distinct orders: *temporally* and *spatially*. Temporal order processes all tiles of a single raster first in order of their index, while spatial order processes the same positioned tiles of all rasters first. Figure 9 shows the order of processing the tiles of a time series with two rasters in both spatial and temporal tile order. The numbers in tiles are the tiles position in the processing order and not their tile index. The figure assumes that the tiles of a raster are loaded in row-major order.

Some operators require a specific order for processing rasters in a high-performance way. For example, the temporal aggregation operator can better process a time series spatially. As the operator has to aggregate equally positioned tiles of multiple successive rasters, it naturally fits the spatial tile order. When both rasters from Figure 9 are aggregated into one raster, tiles 0 and 1, then 2 and 3, and so on would be aggregated. That is precisely the spatial processing order. A convolution operator, on the other hand, requires temporal tile ordering because the calculation of each tile also has to access its adjacent tiles. If an operator tree contains operators that do not support the same tile order, like a temporal aggregation followed by a convolution operator, an operator for changing the order must be inserted in between.

5.3 Descriptors

To avoid loading and processing raster tiles unnecessarily, IPRTS uses *descriptors*. They represent a tile by providing its metadata and a method for loading the tile data. A descriptor contains information about the tile as well as the raster it is part of. The raster information includes its spatial and temporal domain, its total tile count, the tile order, a nodata value, and a data type. The tile information includes its index, resolution, and spatial domain. Both can contain any additional data that might be needed, for example, fields for the minimum and maximum value of the tile.
Descriptors can also contain information that allows certain operators to process the tiles with higher performance. For example, a descriptor can be created that is not representing data but describes a temporal or spatial gap in the raster time series. They can also define that a raster is part of a *recurring* raster time series. Recurring rasters are part of a time series that contains the same rasters multiple times for different temporal intervals. An example of recurring rasters are the datasets from *WorldClim* [7] that provide rasters for the monthly average temperature between 1970 and 2000. Processing the tiles of a recurring raster time series could usually lead to unnecessary and repeated work in some cases. A raster value extraction could, for example, react to a recurring raster time series by only loading each raster once, even when they are accessed in different time intervals.

5.4 Iterator-Based Tile Processing

Operators in LPRTS process the tiles of a raster time series by iterating over them. The operator tree acts as a modular iterator for the result time series. The iterator is created by applying the processing operators on one or more source time series provided by the source operators. A consuming operator actually iterates over those tile descriptors in a loop to process them as needed.
Because of the iterator-based design, the tiles of the output time series are processed by the operator tree successively. When the consuming operator loads an input descriptor, it will be the result of running each previous operator in the operator tree. This would be contrasted

[7] `http://worldclim.org`

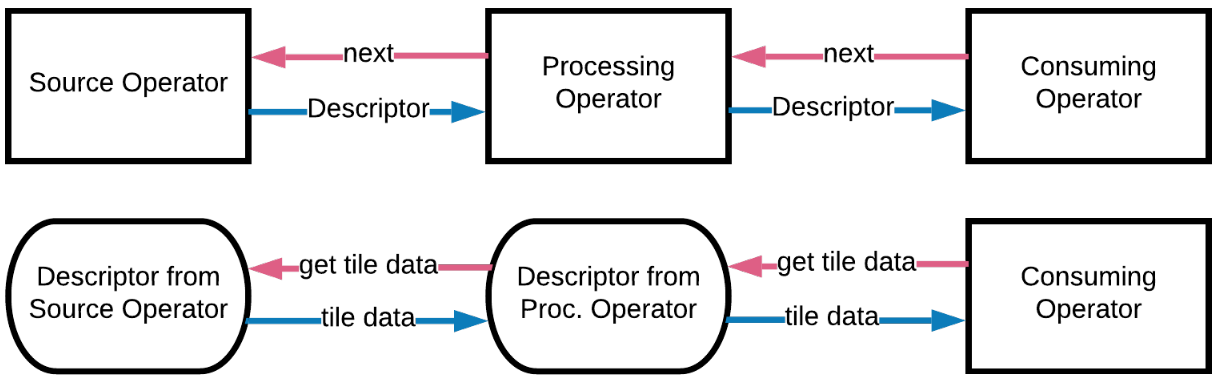


Figure 10: Successive processing of a descriptor and its tile data.

by an approach, that processes a time series operator by operator. In that design, an operator would process all incoming tiles before the system continues with the next operator. The main advantage of the iterator-based approach is that it has less memory overhead. When processing an operator for a complete time series, all of its output descriptors must be kept in memory. Output descriptors also depend on input descriptors to load the input tile data, so potentially all descriptors from multiple operators must be kept in memory. Even though descriptors should have a relatively small memory usage, this will be a significant overhead for very large time series.

To provide the iterator-based and successive processing of tiles, operators have a method for returning the next tile descriptors. Operators that have input operators call their next descriptor method and built their output descriptor based on these input descriptors. The method for creating the tile data of this output descriptor has to load the tile data of the input descriptors and process it as needed. The method of the output descriptor that loads the tile data must load the tile data of the input descriptors. To realize lazy loading, the input tile data should not be loaded before. The sequence of tile descriptors returned by an operator represents a raster time series.

The simplest example is a processing operator that creates one output descriptor for each input descriptor and adds a numeric value to it. More complex is an expression operator that is multiplying two rasters. In the next descriptor method, it has to load a tile descriptor from two input operators. These two descriptors are used to create the output descriptor. When the tile data is returned it loads the data from the input descriptors and multiplies it. Another example is the temporal aggregation operator that loads multiple spatially ordered tiles from one input operator and aggregates their tile data.

The actual tile data is loaded by the consuming iterator when it is iterating over the incoming

descriptors. Due to the descriptors being created successively, the tile data is also processed successively by the whole operator tree. This design avoids unnecessary loading of raster data by realizing lazy loading. Thus sample or filter operators can be implemented without the data of skipped rasters being loaded. Figure 10 demonstrates the successive processing of tile descriptors. It shows an operator tree consisting of three operators. When the consuming operator loads the next descriptor from the processing operator, the processing operator first has to request an input descriptor from the source operator. This input descriptor is used to create the output descriptor by the processing operator. The tile data is loaded afterward by the consuming operator by calling the corresponding method on the input descriptor. As for the descriptor, the processing operator has to first load the input tile data from the source operator to return the tile data.
Because tile descriptors are processed independently, an operator does not have information about the tiles that were created before. But some operators need to keep track of information across multiple created descriptors. For example, a source operator has to keep track of the spatial and temporal domain of the tile it returned last to know which tile has to be returned next. For these cases, operators have to use member variables to keep track of this state.

Tile Skipping and Saving
An essential feature of the iterator-based design is that operators can skip incoming tiles. This can be done by getting the next descriptor from the input operator multiple times and not using the descriptors that are not needed. This way an operator can reduce the number of rasters in a time series.
To keep the result of an operator tree a time series, operators are only allowed to skip complete rasters but not individual tiles of a raster. When an operator returns a tile, all other tiles of the corresponding raster must also be returned. If this is not done, it can not be assumed that the operator works with any combination of different operators. As described before for temporal aggregation, operators can also load multiple input descriptors of a time series to produce a single output descriptor.
Iterating over the tiles of an input operator can only be done forwardly. If an operator has to access an input descriptor again to create the next or another output descriptor, it has to save the descriptor in a member variable and access it accordingly.

5.5 Focal Operations

As IPRTS processes tiles independently, focal operations are a central difficulty for its iterator-based approach. For the edge cells of a tile, a convolution operator needs to access cells of neighboring tiles to calculate the output value. Figure 11 shows an excerpt from a raster. The yellow and red cells are from two different, adjacent tiles. To execute a focal operation on cell c, all the cells in blue outlines have to be accessed, including cells from both tiles. Because operators iterate over each incoming tile independently, they can not access neighboring tiles. This problem is inherent to the iterator-based design of IPRTS. Saving the tiles accessed before is also not enough because the operator might also have to access the following tiles. In the following, four concepts for handling focal operations in IPRTS will be discussed.

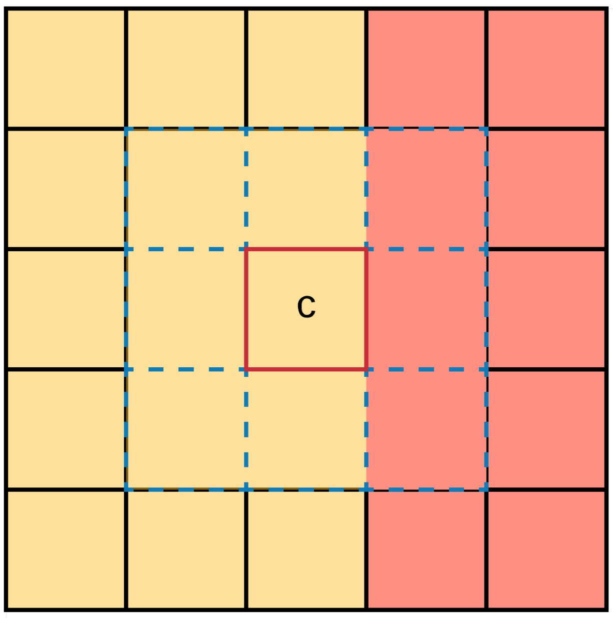


Figure 11: Red and yellow cells are from different tiles. A focal operation on cell c needs to access cells from both tiles.

5.5.1 Concept 1: Plain Iterators

The first concept uses the design as described before. The convolution operator has to detect when it loads the first tile of a raster by its tile index. Then the operator has to load all remaining tile descriptors of the raster and store them in a list. To create the output descriptors of the current raster, the operator has to access the input descriptors as needed from the list instead of loading them from its input operator. This is a workaround that solves the problem of providing focal operations but contradicts the core design of having iterator-based processing of raster time series as it is not processing the descriptors individually. The following sections discuss additions to the design of IPRTS that solve focal operations in different ways.

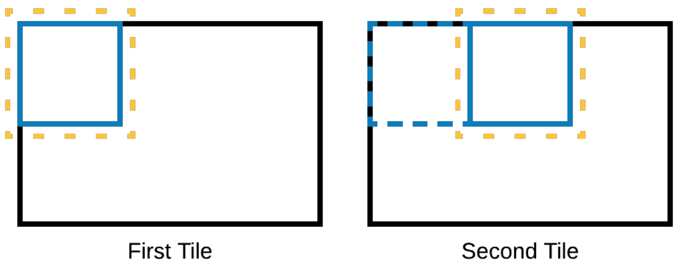


Figure 12: Blue shows the tile and step size, while dotted yellow is the window size.

5.5.2 Concept 2: Step and Window Size

The second concept extends the base design by splitting the tile size into two parameters: the window size and the step size. An operator that needs cells adjacent to the actual tile can adjust the window and step size of their input operators accordingly. A tile with the normal size of 256×256 cells could be adjusted to provide two extra cells at each border by setting the window size to 260×260 cells and keeping the step size at 256×256 cells. Figure 12 depicts this approach with blue rectangles being the step size and the yellow rectangle being the window size. The positioning of the window for the start tile is calculated by subtracting half of the difference between step and window size from the actual tile position. By increasing only the tile size in the base design, this could not be achieved because the actual tiles would not overlap but only be bigger.

One necessary restriction is that an operator has to set the same window size, step size, and query rectangle to all of its input operators. This allows calculations between tiles to be handled easily without comparing tile position and size. But this rule also applies to regular operators when they don't handle it explicitly in a different way.

The most significant advantage of this design is that the focal operators would not have to manage multiple descriptors for a single output tile anymore and thus tiles would only be needed once.

Some raster file formats save the data already tiled. For these cases adjusting the window size could have negative performance effects because loading the tile would require accessing multiple tiles on disk. The incoming tiles could be cached but are not usable for queries that cover the same data but only differ in not having an adjusted window size. Executing multiple convolutions on the same data in a row would result in a window size that is much bigger than the normal tile size for the last convolution operator.

5.5.3 Concept 3: Grid of Tiles

The third concept extends the base design similarly. It also provides a step and window size instead of a single tile size parameter. But the size of tiles that are actually created and processed stays in a system-wide fixed size. The actual size of tiles is decoupled from the query tile size.
Operators still return only one descriptor for the tile but loading the tile data will return a grid of tiles. Every tile overlapping with the query tile size will be part of the grid. The tiles of the grid are the complete, regular tiles that will be created by a descriptor that was loaded before or will be loaded afterward. Because tile data will be accessed multiple times, caching mechanisms should be used. When the query tile size fits the actual tile size, operators will return a 1×1 grid only containing the tile covered by the query.

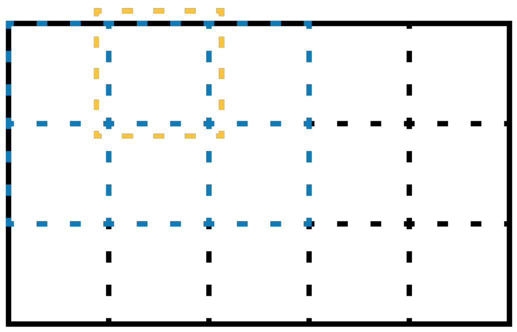


Figure 13: Yellow is the queried tile. Blue are the fixed size tiles that will be returned as a grid. Black are the tiles of the raster that are not returned.

This concept avoids the problems of caching tiles with diverging tile sizes. Operators would have to adjust the handling of processing tile data when loading tile data from a descriptor because they can not assume anymore that only one tile will be returned. Even though operators know if they changed the query tile size themselves, they still have to assume that other operators changed the query rectangle recursively and check every time they load an input descriptor. This requires extra checks for every operator, adding complexity. Other concepts do not have this disadvantage while also fixing the problem.

5.5.4 Concept 4: Random Tile Access

The fourth concept does not change the tile size in any way. It adds *random tile access* to operators. Additional to loading the next descriptor from an operator, any tile descriptor can be loaded. A tile can be requested with its tile index. Random tile access allows choice of

access in the spatial dimensions but not in the temporal dimension. The requested tile will be part of the raster that was returned last by loading the next descriptor.
The difference between iterating access and random access to descriptors is that the iterating access changes the state of input operators. The random tile access does not change how the iterating access returns tile descriptors. Independent from the tile order for iterating, the random access does not allow to load tiles from different rasters of the time series.
Compared to the base design, random tile access creates new requirements operators have to fulfill in IPRTS. When returning a tile descriptor, operators cannot change their spatial-temporal state to the next tile descriptor that will be returned because another tile of the same temporal state might be loaded by random tile access.
With random tile access, focal raster operations can be executed more cleanly in comparison to the base design. Instead of saving all tile descriptors of a raster into a list and accessing them from there, the operator can load and save only the first tile descriptor and access the other tile descriptors as needed with the random tile access. For this use case, operators must additionally provide a function that skips all tiles that are left from the current raster in the iteration.
This concept creates additional functionality that the others don't. The random tiles access allows easier mirroring of rasters and easier changing of the tile order from spatial to temporal order. In the base concept, this change of tile order requires loading all tile descriptors of a time series into a list in the beginning. Afterward, the tiles can be returned in changed order by addressing them with an index. With random tile access, only the first tile descriptor of each raster must be loaded and saved by iterating. The other descriptors of the raster can be loaded with random tile access. Changing the tile order from temporal to spatial is not made easier by this concept.

6 Implementation

Implementing *Iterator-Based Processing of Raster Time Series* lead to many interesting challenges. In the following, problems, decisions, and details of the implementation process will be discussed. IPRTS was developed in C++ with CMake as the build system on Ubuntu 18.04.1. The source code can be found on `github.com` [8]. The implementation uses C++17 features like the standard library additions of `std::optional` and `std::filesystem`. To be able to use these features, `g++ 8` is used as the compiler. For managing time variables, the `date_time` module of the `boost` library is used [9]. To parse queries from JSON format, the `jsoncpp` library is used [10]. `std::filesystem` and `std::optional` could be replaced by `boost::filesystem` and `boost::optional` with small adjustments to avoid depending on a C++17 compiler. A feature from C++14 that is widely used in IPRTS are *lambda capture expressions*.

The main components of IPRTS are the `GenericOperator`, `Descriptor`, and `Raster` classes. `GenericOperator` is the based class for all operators that create and process descriptors of tiles of a raster time series. The `Descriptor` class contains metadata about a tile and provides a method for loading its actual tile data. The tile data is represented in the `Raster` class that allocates memory of a certain size and data type to store the cell grid of a raster or tile.
`GenericOperators` create a `Descriptor` in the `nextDescriptor` method that realizes the iterator-based processing of raster time series. To defer the loading of the actual raster data, operators create a closure that returns a `Raster` and stores it in the descriptor. The closure can be executed from its `Descriptor` by calling its `getRaster` method. In the following, these parts will be explored in more detail.
The implementation of focal operations uses the fourth concept that was discussed in Section 5.5 by implementing random tile access. However, the implementation of random tile access will be explained separately in Section 6.9 and the following will focus on the implementation of IPRTS without it.

6.1 Descriptors

The `Descriptor` class contains multiple data fields for the metadata about the tile it is describing. These are, among others, the spatial-temporal domain and resolution about the raster and tile, the tile index, and the total tile count of the raster. The `getRaster` method executes a `std::function` object that is created by the operator that also created the de-

[8] `https://github.com/SoerenHoffstedt/raster-time-series`
[9] `https://www.boost.org/`
[10] `https://github.com/open-source-parsers/jsoncpp`

scriptor. Generally, a `std::function` is a function wrapper that can store function pointers as well as closures. A closure is a function object that can be executed, that can access variables from its creation context by capturing them, and that is created by lambda expressions. Closures define an anonymous data type that can be executed and stores all captured variables in member variables. Creating a `std::function` from a lambda closure will store the captured variables in dynamic memory. The full data type of the object is `std::function<std::unique_ptr<Raster>(const Descriptor&)>`. This means the function returns a unique pointer to a `Raster` and has a constant reference to a `Descriptor` as a parameter.

When an operator uses incoming tile descriptors and creates its tile data based on the input data, it must capture the input descriptor into the `getRaster` closure. Because the `std::function` object uses dynamically allocated memory to stores captured variables, it is better to move than copy input descriptors. Moving the descriptor will not copy the dynamic memory but move its ownership to the new descriptor. Moving variables into closures is only possible through *lambda capture expressions* that were introduced in C++14. They generally allow assigning named capture variables with an expression in the capture block of the lambda [13]. By capturing `[desc = std::move(desc)]` as used in Listing 1, a descriptor can be moved in the closure. The listing also demonstrates the workflow of loading an input descriptor and creating an output descriptor that processes the input tile data.

The `Descriptor` class inherits from the `DescriptorInfo` class purely for usability reasons. In many operators, the meta information does not change at all or at least very little between the input and output descriptor. Instead of copying all the fields one by one, the classes are separated to allow easy copying. `DescriptorInfo` contains all the metadata about the tile, while `Descriptor` contains the `std::function` for saving the `getRaster` closure. Due to this separation, the metadata of a tile can be copied and passed into the constructor of the output descriptor easily, as shown in Listing 1. When assigning `desc.value()` to `info` all the fields of `DescriptorInfo` are copied from the input descriptor. The `value` method must be called on `desc` because it is an `std::optional<Descriptor>` and returns the raw object (more on this in Section 6.6).

Descriptors also contain a special field that indicates if their tile only contains nodata values. This allows for optimizations in processing operator that can skip calculation on this descriptor. Nodata descriptors can, for example, be used when a query processes multiple raster time series and its spatial domain is bigger than the extent of one of the time series. When a complete tile is outside of the raster's spatial domain, they can be created as nodata descriptors. The descriptor class provides a static function for creating a nodata descriptor. It creates a `getRaster` closure that fills all cells of the tile with the nodata value.

Code 1: Copying the metadata of an input descriptor.

```
auto desc = input_operators[0]->nextDescriptor();
DescriptorInfo info = desc.value();
auto getter = [desc = std::move(desc)](const Descriptor &self)
-> UniqueDescriptor {
  auto input = desc->getRaster();
  auto output = Raster::createRaster(...);
  //Work with input and output raster.
  return output;
};
return std::make_optional<Descriptor>(std::move(getter), in);
```

6.2 Operator Tree

The `OperatorTree` is a recursive data structure that represents a logical view of an operator and its input operators. Each node contains the operator name, parameters, and the query rectangle. Additionally, a node points to its input operator trees that represent its input operators. An `OperatorTree` is built from a query in JSON format. Each operator tree node can instantiate an operator and its input operators by creating the actual operators used for processing a raster time series. Operators are allocated as unique pointers to handle them polymorphic as a `GenericOperator`.
Each `GenericOperator` keeps a reference to its `OperatorTree`, allowing for query subsetting and cloning of operators. When re-instantiating an operator, it is possible the change the query rectangle to, for example, split a query into multiple parts.
The implementation of every operator in IPRTS inherits from the `GenericOperator` class. It primarily provides the virtual methods `nextDescriptor` and `initialize` to implement the operators functionality.

6.3 Initialization

After instantiation of an operator tree, an initialization function is called that recursively calls the `initialize` method of each operator, starting from the consuming operator. In this step, each operator has the chance to initialize internal variables and to adjust the query rectangle or parameters of its input operators. An example that requires changing the query rectangle of its input operators is the `OrderChanger` operator. It changes the order of the incoming tiles, and thus it sets the order in the query rectangle of its input operators and their children to the opposite.
Adjusting the query rectangle and parameters of input operators could not be done in the constructor of an operator because the operators are instantiated bottom up. Initializing oper-

ators must be done top down because operators might initialize their input operators differently based on the changes from its parent operator.

6.4 Consuming Operators

Consuming operators are the root of an operator tree as they don't produce output tile descriptors but only consume them. All implementations of consuming operators inherit the `ConsumingOperator` class that itself is a subclass of `GenericOperator`. It overrides the `nextDescriptor` method to return nothing and additionally provides the virtual `consume` method.
In the `consume` method the operator iterates over the incoming tile descriptors and processes them as desired. The processing cannot be done in `nextDescriptor` because consuming operators do not return descriptors themselves. `consume` is a unified interface to execute all types of consuming operators after instantiating an operator tree.
For iterating the tile descriptors of its input operator, the `ConsumingOperator` can use a C++ for-each loop. `GenericOperator` implements the `begin` and `end` methods that returns objects of the `TimeSeriesIterator` class. It implements a C++ forward iterator, keeps a pointer to the operator it was created from, and stores the descriptors it is loading in a member variable. When instantiating an operator tree from a JSON query, the consuming operator can be instantiated in a separate method returning a `std::unique_ptr<ConsumingOperator>`. The query can be executed by calling `consume` on this operator. Figure 14 demonstrates in a sequence diagram the execution of an export operator. It shows how tiles are processed successively in IPRTS.
Two consuming iterators were implemented for IPRTS. The fist is the *Raster Value Extraction* that will be discussed in Section 6.10.4. The second is the *GeoTiff Exporter* that exports each raster of the time series created by the query to disk and will be discussed in Section 6.10.2.

6.5 Source Operators

The `SourceOperator` class implements common functionality for all source operators. For example, it implements the advancing of the spatial and temporal state of the operator that defines for which tile a descriptor is created. Source operators primarily have to implement the `createDescriptor(double time, int pixelStartX, int pixelStartY)` method. It must return a descriptor for the raster starting at `time` and its tile starting at the pixel position defined by `pixelStartX` and `pixelStartY`. `SourceOperator` handles increasing the time and pixel state and the actual source operators only have to implement the actual creating of descriptors and loading of tile data.
Different storage types, interfaces, or libraries can be used as the back end of a source operator.

A requirement for a back end to support tiles is that it can load any rectangle of pixel data into memory and not only the complete raster. Additionally, it must support the data layout used in IPRTS. The cells of a raster are stored in row-major order, meaning that each grid line is stored in a sequence starting at the top. For IPRTS a source operator based on the GDAL library is implemented (see Section 6.10.1). For debugging purposes, it additionally provides the `FakeSource` operator the fills rasters with the same unique raster index instead of real data.

6.6 Unique or Optional Descriptors

The iterator-based design of the operators is built around the `nextDescriptor` function. The consuming operator of an operator tree iterates over the incoming tiles by calling `nextDescriptor` until the function communicates that no further tiles are available and the end of the time series is reached. In the following, it will be discussed how operators return

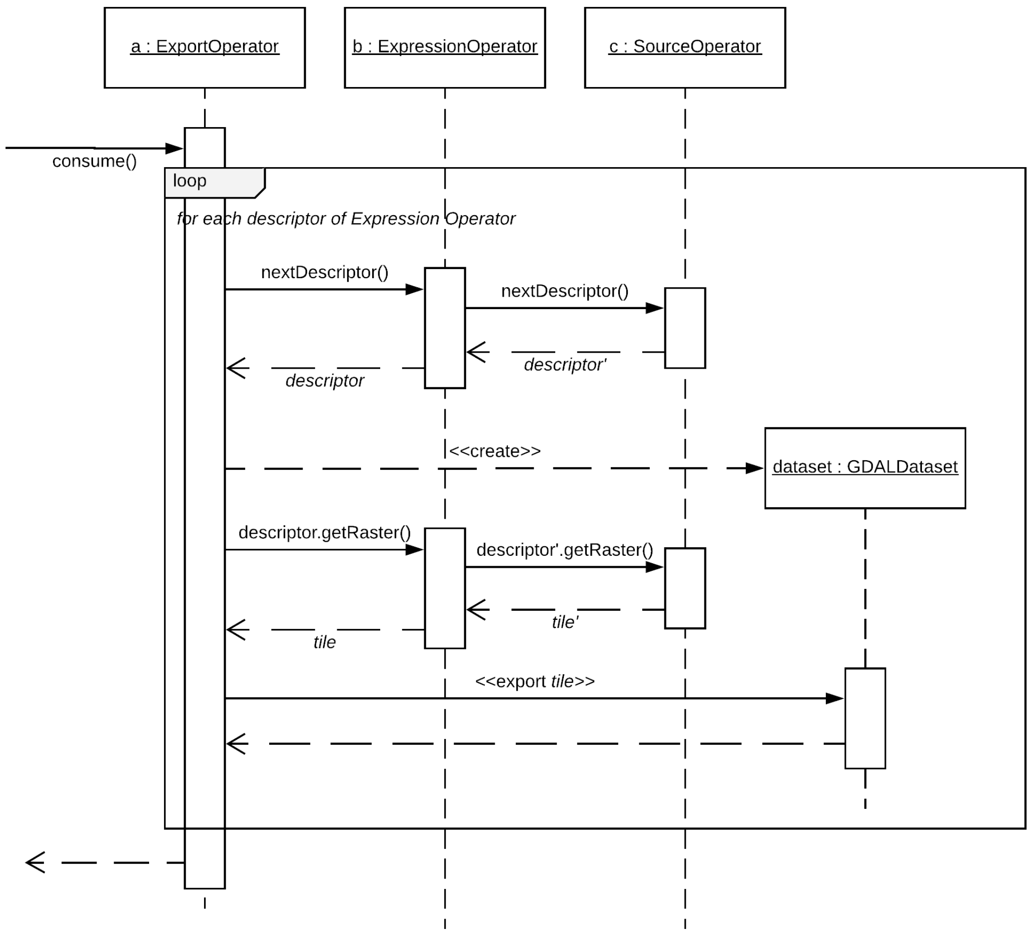


Figure 14: Sequence diagram of a consuming operator processing tiles by iterating over them.

objects of the `Descriptor` class.
Because IPRTS is implemented in C++, it could be assumed that the operators are C++ iterators. An iterable data structure in C++ is not the iterator itself but creates an iterator object that points to its data [14]. For example, using a data structure in a C++ for-each loop requires it to implement `begin` and `end` methods that return iterator objects [15].
Iterating over the tile descriptors of an operator is more like the iterator concept of Java. A data structure in Java that can be iterated must implement an interface providing the `hasNext` and `next` methods. It separates checking for the availability of a next object in the `hasNext` method from returning the object in the `next` method. The `nextDescriptor` design in IPRTS works similar to these Java iterators whose `next` and `hasNext` methods could also be used to chain iterators. But IPRTS does not split between `next` and `hasNext`. Instead the `nextDescriptor` indicates that no further descriptors can be created by returning a not valid object.
Two possibilities for indicating non-valid objects exist. The first dynamically allocates descriptors and returns a `nullptr` when the end of the time series is reached. For memory safety and code cleanliness a `std::unique_ptr<Descriptor>` should be used instead of a raw pointer. The second returns descriptors as stack-allocated objects but wraps them in an `std::optional` object. Some operators require that a `Descriptor` is copyable because it must be accessed in multiple consecutive `nextDescriptor` calls. Examples will be presented in Section 6.10.
Unique pointers are not copyable as they uniquely own the memory they point to. A class could provide a cloning function that allocates a new object, copies all its data fields into it, and returns it as a unique pointer. In the case of a `Descriptor` this is not possible due to the `std::function` object. Generally, `std::function` objects can be copied but in many use cases operators will move descriptors into its closure. Therefore, the `std::function` contains a descriptor that cannot be copied but must be cloned and thus returning descriptors as unique pointers is not possible.
During development it was also explored to store the `getRaster` closure as the anonymous type created by a lambda expression in a subclass of `Descriptor`. In this case, descriptors would have to be returned as a unique pointer because a descriptor variable must be polymorphic. In this implementation, cloning descriptors was also not possible because the anonymous type of a closure is not copyable [13].
`std::optional` is a stack allocated object wrapper that either contains a value or not, and is copyable. `std::optional<T>` objects can be assigned with the `std::make_optional<T>(...)` function and objects without a value can be assigned with the `std::nullopt` constant. The object stored by a `std::optional<T>` can be accessed by dereferencing it like a pointer or calling its `value` method. Because optional descriptors contain dynamic memory in the `std::function` object, they should also be moved into closures. Most of the data fields in the descriptor will still be copied when the complete object is moved because it is a stack allocated object.

6.7 Tile and Raster Skipping

In the iterator-based design, operators can skip single incoming tiles or complete rasters by loading their descriptors but not using them. To skip a complete raster in temporal order, an operator can call `nextDescriptor` on its input operator in a loop based on the tile index and the tile count of the raster as shown in Listing 2.
Even though no raster data is loaded for the unused descriptors, it is still an overhead to create these descriptors, especially the longer the operator chain is that created them. To allow tile and raster skipping without this overhead, `GenericOperator` provides the methods `skipCurrentRaster` and `skipCurrentTile`. The standard implementation just calls the method recursively on all input operators and is overwritten by `SourceOperator`. As `SourceOperator` handles the temporal and spatial state about the descriptor that was last returned, calling these skipping methods will increase the state accordingly. Both functions can be called in each tile processing order, but operators that use them must make sure that their output time series only contains complete rasters. To skip a raster, equivalent to the code in Listing 2, `skipCurrentRaster` must be called once.
In temporal order, `skipCurrentRaster` skips all remaining tiles of the current raster. The temporal state of the source operator is advanced to the next raster of the time series, and the spatial state is reset to the first tile. In spatial order, only the temporal state is advanced to the next raster. The spatial state is only advanced when the last raster of the time series was skipped.
In temporal order, `skipCurrentTile` will skip a single tile of the current raster by advancing the spatial operator state. When the last tile of a raster was skipped, the temporal state is advanced to the next raster of the time series. In spatial order, the current tile is skipped for all remaining rasters of the time series by advancing the spatial state and resetting the temporal state to the first raster of the time series or the starting time of the query.
Both functions allow skipping more than one raster or tile at a time by providing a number of skips as a parameter. For spatial order, it is important to note that the number of skips will not be completely executed when the last raster of the time series was skipped. The spatial

Code 2: Skipping descriptors of a raster by calling `nextDescriptor()` multiple times.

```
auto inputDesc = input_operators[0]->nextDescriptor();
int end = inputDesc->rasterTileCount;
for(int i = inputDesc->tileIndex; i < end; ++i){
  input_operator[0]->nextDescriptor();
}
//load first tile of next raster:
inputDesc = input_operator[0]->nextDescriptor();
```

state of the operator will be advanced to the next tile and no rasters of this tile will be skipped. This is necessary because operators do not know how many rasters a time series contains in total.
For operators that change the temporal resolution of a time series, the implementation is more complicated, so they must overwrite the methods accordingly. For example, the *temporal aggregation* operator cannot simply skip one raster of its input operators but must skip all incoming rasters that would be aggregated to its next output raster. Based on its aggregation time interval (see Section 6.10.5, the operator can calculate the time interval of the next aggregated raster that must be returned. Then the operator has to load a descriptor from the input operator and check if it or the next descriptor falls into the time interval. If not, the operator skips one raster on the input operator and does the same check again for the next descriptor.
When the last raster was skipped in spatial order, the spatial domain of the input operator will be advanced, and the temporal state will be reset to the first raster of the query time series. The operator cannot detect this reliably before calling skip because it is not known how long the time series actually is. In these cases, the operator has to save the loaded descriptor in a member variable. If the variable contains a value, the operators `nextDescriptor` method will use it as an input instead of loading the next descriptor. In most cases, it can be seen by the end time of the loaded descriptor that the next descriptor will be in the interval and this variable does not have to be used.

6.8 Raster

The raster data created by the `getRaster` closure of descriptors is returned as a unique pointer to an object of the `Raster` class. It dynamically allocates memory for the tile data and additionally contains fields for the tiles resolution and data type. Rasters can be of different data types like `byte`, `uint16`, `int32`, or `float`. They range in size from 8 to 64 bit, can be signed or unsigned, and can be integer or floating point based.
The data pointer is part of the template class `TypedRaster<T>` that inherits from `Raster`. It also provides all data type dependent functionality like returning its size or value range, reading from cells, and writing into cells. The class also contains methods for directly accessing the data pointer, as `T*` or `void*`. As tiles can only be partially covered by actual data (see 5.2), the data pointer can also be returned with an offset. This way, a pointer to the beginning of the actual relevant data can be obtained.
Rasters can be created by the static `createRaster` method that returns a unique pointer to a `Raster` and allocates the correct `TypedRaster<T>` objects based on the passed data type.
`Raster` objects can handle the memory for the tile data in two ways. First, they allocated and own the memory for the tile data themselves and when the `Raster` object is destructed, the memory of the data pointer is also deallocated. Second, the `Raster` gets a data pointer passed

Code 3: Deducing the underlying type of a raster pointer by its data type.

```
Raster *raster = ...;
switch (raster->getDataType()) {
    case GDT_Byte:
        return (TypedRaster<uint8_t> *)raster->setCell(x, y, 5);
    case GDT_Float32:
        return (TypedRaster<float> *)raster->setCell(x, y, 5.5);
    case GDT_Int16:
        ...
}
```

to the constructor and does not own its memory. On destruction, the data pointer will not be freed. This allows to cache tile data and to return it multiple times without recreating the same data.

As the data type of `Raster` objects can differ, writing values into cells is implemented in the `TypedRaster<T>` subclass. This makes it difficult to use the base class `Raster` when processing tile data. To read and write with the right data types of a class, the `Raster` has to be cast to its actual type: `TypedRaster<T>` with `T` being the raster's data type. This cast can be executed with a switch statement as shown in Listing 3.

This solution becomes more complicated when multiple rasters are involved that might have different data types. It would also result in duplication of the type casting code in most operators. Therefore, IPRTS provides static template functions in the `RasterOperations` class that execute operations on rasters and handle these casts. It provides methods for operating on one, two, or three rasters. They are called `callUnary`, `callBinary`, and `callTernary`. The operation must be implemented as a separate template class that provides a `rasterOperation` method. The method can have a variable amount of parameters. Based on the number of raster operands, the first one, two, or three parameters of the method are pointers to `TypedRaster<T>`. To be usable by the `RasterOperations` class, each raster must have its own template type even when the rasters might have the same data type in the actual use case. The `rasterOperation` method can take any number of additional parameters as the operation needs them. Listing 4 demonstrates the implementation of the raster operation struct `AllValueSetter`. It will set each cell of a raster to a value.

Listing 5 shows how the `callUnary` method of the `RasterOperations` is implemented. The nested template parameter `function` is the raster operation class or struct that implements the `rasterOperation` method. As in Listing 3, the method switches over the data type of the raster and casts its pointer to the corresponding `TypedRaster<T>` type. Additionally, it calls the `rasterOperation` function of the `function` template parameter. Because it is a nested template parameter, the deduced data type of the raster must also be provided as a

Code 4: Implementation of a raster operation class.

```
template<class T>
struct AllValuesSetter {
    static void rasterOperation(TypedRaster <T> *raster, T val) {
        Resolution res = raster->getResolution();
        for (int y = 0; y < res.resY; ++y) {
            for (int x = 0; x < res.resX; ++x) {
                raster->setCell(x, y, val);
            }
        }
    }
};
```

Code 5: Generic method to execute unary raster operations that handles the raster type deduction. Part of the `RasterCalculations` class.

```
template<template<typename T> class function, typename... V>
static auto callUnary(Raster *raster, V... v)
-> decltype(function<uint8_t>::rasterOperation(nullptr, v...)){
    switch (raster->getDataType()) {
        case GDT_Byte:
            return function<uint8_t>::rasterOperation(
              (TypedRaster<uint8_t> *) raster, v...);
        case GDT_UInt16:
            return function<uint16_t>::rasterOperation(
              (TypedRaster<uint16_t> *) raster, v...);
        ...
    }
}
```

template argument to `function`. The casted raster and the other parameters are passed to the `rasterOperation` call. To execute the `AllValueSetter` to set every cell of a raster to 5.5, `RasterOperations::callUnary<AllValueSetter>(rasterPtr, 5.5)` must be called.

6.9 Random Tile Access

Following the discussions of different concepts for implementing focal operations in IPRTS in Section 5.5, it was decided to implement the fourth concept that expands the iterator-based design of IPRTS by providing random tile access. Concept four was chosen because the random access does not only allow the implementation of a convolution operator but also provides additional possibilities. Especially, its usage in an order changing operator is interesting. This and other use cases will be compared in benchmarks to the implementation of concept one in Section 8.1. Concept one only uses tile iteration and stores tile descriptors in lists to provide convolution and order changing.
Random tile access is added the `GenericOperator` class with the `getDescriptor(int tileIndex)` method. Just like `nextDescriptor`, the method returns a `std::optional<Descriptor>`. To support different tile access patterns like z-curve, the method takes a one-dimensional index as a parameter to identify the requested tile. It represents the position of the requested tile in the iteration order of the tiles.
Generally, adding random tile access to operators might require restructuring operators. To minimize code duplication, the code to generate an output descriptor should be moved into a separate function callable for both `nextDescriptor` and `getDescriptor`. With operators having state variables to transfer information between consecutive `nextDescriptor` calls, this function should not change state variables that would affect a `nextDescriptor` call. Source operators are a special case, and their base class `SourceOperator` implements the functionality for both functions, so that source operators only have to provide one function for both cases. In `getDescriptor` `SourceOperator` will calculate the spatial coordinates and pixel positions of the requested tile based on the query rectangle. This means that `getDescriptor` requires additional calculations in comparison to the normal iteration.
When a tile is requested from an operator by calling `getDescriptor`, the raster it is part will be based on the temporal state of the operator. The temporal state of the operator is set by its last `nextDescriptor` call. For this reason, the concept is called random *tile* access as it does not provide random access to any raster. Operators have to take into account that calling the methods for skipping tiles or rasters will change the temporal and spatial state of the operators.

Problem of Temporal Changes to the Time Series
The random tile access is a challenge for operators that change the temporal resolution of a time series. Operators like *Temporal Aggregation* require access to multiple rasters to calculate the output raster. The temporal state of its input operator will be set to the last raster loaded from it. To provide random tile access, the temporal aggregation operator needs to be able to access rasters at different times.
To solve this, it has to use of re-instantiating operators with a sub-query. The temporal

aggregation operator has information about the time interval it was aggregating (see Section 6.10.5). By taking this time interval and calculating the spatial information of the requested tile index, the operator can clone its input operator and pass it a query rectangle with this spatial-temporal domain. Afterward, it can iterate over all incoming tile descriptors of the cloned operator with the `nextDescriptor` method. After cloning an operator, it is important that its incoming tiles will a have different spatial domain for the complete raster than the query rectangle of the operator. Depending on the use of the tile descriptors, the operator might have to change this information to the expected values.

The *sampler* is another operator that changes the temporal resolution of a time series and provides challenges in the context of random tile access. Even though the problem is not related to its `getDescriptor` implementation. It allows skipping of rasters to lower the amount of processed data. For example, it can skip every second raster of a time series.

To keep its output time series without gaps, the operator has to extend the temporal validity of the returned raster tiles to also cover the time interval of the rasters that are skipped next. That the raster time series is without any gaps is an important feature for different operators. As operators cannot advance their temporal state in `nextDescriptor` to the time of the following raster, they cannot load and store the input next tile descriptor to use it in the next iteration step. The temporal aggregation operator, for example, has to identify the last raster of the current aggregation interval based on the end of its temporal domain. As the operator maps rasters into aggregation intervals based on their starting time and not their complete temporal validity, this is possible. When the validity of the current descriptor ends outside of the aggregation interval, the next descriptor will not be part of it. This would not be possible if a raster time series could contain temporal gaps because the operator can not identify before loading a descriptor that it will not be part of the current aggregation interval. But the sampling operator cannot know the start time of the raster it will return next. Raster time series in IPRTS can be irregular. Even when all time series that can be used as input time series would be regular, applying the temporal overlap operator often creates an output time series with irregular time patterns. Thus, the operator can not simply calculate the next start time. The sampling operator also cannot load the skipped tiles once because it would alter the temporal state of its input operator and random tile access would not be possible.

As sampling is an important feature in processing raster time series, a solution for this problem can be found in the descriptor concept. To support sampling, a boolean field can be added to the descriptors that symbolizes that the tile is skipped and the `getRaster` closure can return a null pointer. Most operators would have to check for "skipped" raster to avoid work on a tile descriptor that does not describe a real tile. This is obviously an overhead but it would make random tile access and time series sampling possible.

General Observations
The addition of random access is a departure from the iterator design of IPRTS the same way as the workaround described in Section 5.5.1. But providing an interface to accessing other tile descriptors is more structured than every operator loading all descriptors of a raster into a list when needed. A potential advantage is better order changing from spatial to temporal tile order. Even though the random access of tiles to the current raster moves IPRTS away from a pure iterator-based design, it still means that the design is an iterator in its core. Even when `getDescriptor` is used, the sequence of accessing complete rasters does not change. It is an additional feature more than a complete change of the approach.
Especially for a convolution operator, random tile access requires tile data caching to allow efficient access the same tile multiple times. A tile caching solution is presented in Section 6.10.8.

6.10 Proof of Concept Operators

The operators discussed in Section 2.3 represent basic features a flexible system for processing raster time series must fulfill. All of them are implemented in IPRTS and interesting implementation aspects will be discussed in the following. Additionally, the implementation of a source and a GeoTiff export operator based on the GDAL library as well as complementary operators will be discussed.

6.10.1 GDAL Source

The GDAL is an open source C++ library that allows reading and writing most common raster data formats. It abstracts their differences by providing uniform API to use them. The primary source operator implemented in IPRTS is `GDALSource` and utilizes the strengths and flexibility of GDAL. It supports loading rasters from GeoTiff files on disk, from WCS/WMS requests to web servers, from PostGIS databases, and more. Another advantage of using GDAL is that it supports features like overview files to GeoTiff rasters. These files contain the same raster information at multiple smaller resolutions. Storing them makes queries at smaller resolutions faster because the raster data does not have to be interpolated at runtime. GDAL can also translate the projection of a raster and handles interpolation of requested raster data when the requested resolution does not match the source data.
The main use case for `GDALSource` are raster time series that are stored as separate files on disk. The rasters are each valid for a regular time interval, like three month, one day, or five hours. Queries request a data set in the operator parameters. The data sets are defined by a JSON file on disk and contain a start and end time, a time interval, the locations of the files, and a mapping from a time to a filename. This way the operator can increase the time of the

last opened raster to the next raster and by inserting the time into the filename it can access the raster file on disk.

6.10.2 GeoTiff Exporter

GeoTiff is a commonly used file format for raster data and therefore, exporting the rasters of a time series resulting from an IPRTS query is an important operator. The operator also utilized the GDAL library for creating and writing data into GeoTiff files.
For the first tile of a raster, the operator creates a new file by creating a new `GDALDataset`. For temporal tile order only one `GDALDataset` must be open at a time, but for spatial tile order, multiple are kept open in a list. The maximum amount of open datasets is defined by a constant and it should be tested with big raster time series how many datasets can be kept open. When the maximum is exceeded, datasets must be closed after writing and be opened again when another tile must be written into it. Each tile is written into the file separately. For the edge tiles of the time series, the operator must calculate how many pixels from the tile must be written into the file because they potentially cover more space than the complete raster.

6.10.3 Expression

The `ExpressionOperator` provides local arithmetic operations on one or two raster time series. The current implementation allows for two operands and one operator in an expression. An operand can be a raster or a numeric value. The currently supported operators are addition, subtraction, division, multiplication, and modulo. The expression is defined as a string in the query parameters of the operator. Rasters are represented as capital letters with A being the first and B being the second input raster of the operator. Example expression are $A + B$, or $A \cdot 3$.
Expression with more than two operators can be executed by nesting multiple expression operators. The operator could be extended in future development to support an arbitrary amount of operands and operators.
The functionality of parsing and executing the expression is implemented in a separate `Expression` class that can be used by multiple operators as needed. The `Expression` class provides a method for creating the `getRaster` closure for an expression. The input tile descriptors for the closure are passed by the `ExpressionOperator` as a list.
For expressions on two raster time series, the operator does not change or match the rasters from the different time series onto each other. Instead, it just processes the rasters in the order as they come in even when their temporal validity is not the same. The spatial domain and tile size of both inputs will match because Because both time series are based on the same query, their spatial domain and tile size matches.

6.10.4 Raster Value Extraction

Raster value extraction is a consuming operator that returns the cell values for a list of points. The points contain a position in coordinate space of the rasters and a time. They are passed as parameters to the operator that can process them both in temporal and spatial order.
During initialization, the operator sorts the points depending on the tile processing order. The operator is looking for the pixel values of each point sequentially by iterating over the list of points. Thus, the operator does not iterate over the descriptor of the input operator in a loop but loads the next descriptor when needed. Therefore, it compares the spatial and temporal information of the tile descriptor with the point and decides if and how the tile has to be skipped. The operator can make use of the methods for skipping rasters and tiles (see Section 6.7). As an example, when in temporal order the searched point is temporally not part of the raster of the current descriptor, the operator can call `skipCurrentRaster` and proceed with the next descriptor.
For now, the cell values can either be exported in a text document or printed to the console. As the operator is an interface between raster and vector data, future development must integrate the input and output of the operator into the vector data concept of VAT.

6.10.5 Temporal Aggregation

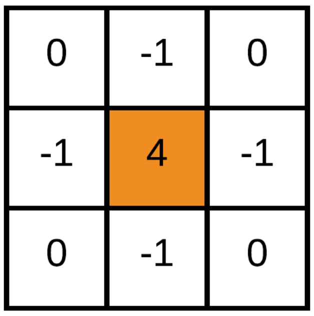
The `TemporalAggregator` operator implements temporal aggregation as described in Section 2.3.3. It supports mean, minimum, maximum, and sum as aggregation functions. They all work on a cell basis as local operations. That means aggregation rasters with the maximum function will create a raster where each cell contains the local maximum of the same cell from the input rasters. The aggregation time interval is defined in the parameters of the operator by a time unit (year, month, day, etc.) and a length. A length of three and month as the unit represent a time interval of three months. The first interval starts at *t1* of the query rectangle. When no time interval is provided in the parameters, the complete time series will be aggregated into a single raster. Input rasters are assigned to an aggregation time interval by their start time. This means a raster could overlap with multiple aggregation intervals but will be assigned only to one.
Temporal Aggregation is a prime example of an operator that benefits from processing tiles in spatial order. The operator calculates what the time interval for the next aggregated tile is and loads input descriptors as long as the next will also start during this time interval. In temporal order, this would be more complicated because other tiles of the rasters would be loaded before the next tile of the same spatial domain. Therefore, for each tile of a raster, the operator would need to save descriptors in a list to support temporal ordering.

6.10.6 Temporal Overlap

Similar to temporal aggregation the `TemporalOverlap` operator has to deal with the complexities of changing the number of rasters in a time series. Instead of lowering it, like the `TemporalAggregator`, the operator increases the number of rasters by temporally mapping two time series onto each other as described in Section 2.3.4.
The operator loads tile descriptors from both time series and compares them temporally. When they don't overlap, it skips the raster of the earlier descriptor until an overlap is found. The output raster is temporally defined by the higher starting time and the lower ending time of both. When one of the rasters is valid longer than the other, it must be saved as input for the next descriptor because it will overlap with the next raster of the other time series (compare with Figure 5 on page 12).
The outgoing raster is defined by an expression as described in the expression operator. Because a raster can be used in multiple overlaps, processing speed will be improved by caching the tile data of a raster. The operator is currently implemented only supporting temporal tile order, changing it to support spatial order is possible though.

6.10.7 Convolution - Edge Detection

The focal operations are a significant problem for the iterator-based design of IPRTS. Operators don't have access to descriptors of neighboring tiles as they process each tile independently. But the operator needs to read cells from other tiles to calculate the results for edge cells of a tile. The `Convolution` operator provides focal operations. As multiple tiles of the same raster must be accessed for each output tile, it requires a temporal tile order. To avoid loading the same tile data from disk multiple times, a cache operator must be used before the convolution operator (see Section 6.10.8). Currently, the operator has one convolution function implemented. It uses a Laplacian 3×3 edge detection kernel [16] (see Figure 15) but any kernel could be implemented



0	-1	0
-1	4	-1
0	-1	0

Figure 15: Laplacian 3×3 edge detection kernel used in Convolution operator.

Code 6: Saving raster tiles to a list, use list as input

```
if(descriptorList.empty() || currTile == rasterTileCount){
  descriptorList.clear();
  OptionalDescriptor desc = inputOperators[0]->nextDescriptor();
  if(desc == std::optional)
    return std::optional; //end of time series reached
  descriptorList.push_back(desc);
  while(desc.tileIndex < desc.rasterTotalTileCount - 1){
    desc = inputOperators[0]->nextDescriptor();
    descriptorList.push_back(desc);
  }
  currTile = 0;
}
OptionalDescriptor inputDesc = descriptorList[currTile];
currTile += 1;
// work with inputDesc...
```

in future development by abstracting the operator's kernel concept.
Two ways of implementing convolution functionality will be discussed. The first way works without random tile access by storing all tile descriptors of a raster in a member variable. When the operator loads the first tile of a new raster in `nextDescriptor`, it immediately loads the remaining input descriptors of that raster and stores them in a list. This list is afterward used to access the tile descriptors and data as needed. Listing 6 demonstrates this strategy. The field `int currTile` indicates which tile of the raster is currently returned. Based on this index, the adjacent tiles must be identified from the `descriptorList`
The raster operation that creates the output tile data for the convolution operator takes a separate list of tiles as input. It contains pointers to their tile data. The tile that is currently worked on is positioned at index 0 and all neighbor tiles starting with the tile above are positioned clockwise from index 1. As edge tiles don't have neighbors in every direction, some spots will be filled with null pointers.
The raster operation iterates two-dimensionally over all cell indices of the tile and accesses the neighbor cells as described by the kernel. Reading from cells is done in a separate method that allows passing negative indices or indices that are bigger than the tile size. For these cases, the method identifies which tile from the list the value must be read from.
The second implementation makes use of the random tile access provided by the `getDescriptor` function. The raster operations do not change compared to the first implementation. Only the creation of the list of adjacent tiles that will be passed to the raster operation changes. Instead of loading and saving all tiles of a raster into a list, the operator loads the tile that has to be processed normally by calling `nextDescriptor` once. The neighboring tiles will be loaded as

needed by calling `getDescriptor` but will not be saved in a list. This takes away the state handling in the convolution operator and allows to handle the creation of each output tile independently.

6.10.8 Other Operators

Sampler

Sampling a time series as described in Section 3 is an important part of processing raster time series. Sampling means to select x number of rasters of a time series. Based on x the operator must know which rasters must be loaded. Operators do not know how many rasters the time series they produce contain in total and therefore real sampling cannot be implemented at the moment. The dataset model behind the `GDALSource` operator can know the number of rasters. But it must be explored in the future if this is true for every type of source operator that might be used for a system like IPRTS. Additionally, it must be tested if the operators that change a time series temporally and especially the temporal overlap operator can calculate the number of time series they provide. Sampling x rasters from a time series with a regular time interval would require to know the total number of rasters.
Therefore, the `Sampler` operator only provides raster *skipping* at the moment. The operator works based on two parameters: the number of rasters to skip and the number of rasters to keep. When both numbers are two, the operator will repeatedly return two rasters and then skip two rasters.

Cumulative Sum

The `CumulativeSum` operator has a similar structure of processing data as the aggregation operator. It is summing all rasters together, but instead of producing one aggregated raster in the end, every input raster also produces an output raster: the partial sum of all rasters to this point. The last output raster is the sum of all rasters of the time series. The operator is separate from the `TemporalAggregator` because it needs to save the current state of the partial sum as a member variable.

Order Changer

As important operators like `Convolution` and `TemporalAggregation` require a different tile order, they can not be used in the same query. The `OrderChanger` changes the order of an incoming time series to the opposite and thus allows to, for example, use a time series created by temporal aggregation as input to the convolution operator.
Without random tile access, the operator has to load all descriptors of the incoming time series and store them in a list at the beginning of the first `nextDescriptor` call. Afterward, the operator uses a raster and tile index to keep track of the tile that has to be returned and

increases them accordingly. Tile descriptors are returned from the list with these indices by calculating the position the tile has in the list.
For large time series, this approach can lead to high memory usage as the size of each descriptor is 264 bytes (plus the size of the variables captures in the `getRaster` closure). The operator can make use of random tile access to change the order of a time series from *spatial* to *temporal* without the additional memory overhead. It can load the first tile of each raster, store it as a single member variable, and return the other tiles of its raster by calling `getDescriptor`. Changing the tile order from *temporal* to *spatial* still requires storing all tile descriptors in a list and is not changed by random tile access.

Raster Cache

As the focal operations like convolution require accessing each tile of a raster multiple times, it makes sense to cache the tile data. The `RasterCache` provides caching to the tile data of a temporally ordered raster on operator level and prevents creating the same tile data multiple times. The operator should, for example, be used before the convolution operator to improve its performance.
It makes use of `Raster` objects that do not own their data pointer. When loading an input tile descriptor, the operator does not change the metadata but changes the `getRaster` closure. The operator stores the `Raster` objects in a list at the position of their tile index. In the closure, the operator checks if the tile is already available in this list. If it is, it will be returned from the list. Else the tile data is loaded from the input descriptor and stored in the list. When a descriptor from a new raster is loaded, the cache is reset by deleting all cached tiles from the previous raster.

7 Example Query

This section presents an example query to IPRTS including visualizations the input and output rasters. Queries are formulated in JSON format and contain two parts.
Because of its length, the query file is split into two listings in this section. Listing 7 defines the query rectangle as a JSON object in line 2. The query rectangle describes the spatial-temporal domain that the output raster time series is requested for. It has five components: the resolution of the rasters (line 3), a temporal reference defining the time interval of the time series (line 6), a spatial reference defining the spatial domain of the rasters (line 9), the processing order of the tiles (line 13), and the resolution of the tiles (line 14). Line 18 indicates that the definition of the operator tree is part of the same JSON file.
The second part defines the operator tree and is shown in Listing 8. Each operator is defined by three parts: the operator name, its parameters, and its input operators. Line 3 defines the operator name of the consuming operator of the query that is, in this case, the `GeoTiffExporter`. In line 4 a JSON object defines the parameters that can contain any data needed by the operator. The input operators of the operator are defined in line 8 by a JSON array. It can contain any number of operator definitions. In this case, the exporter has one input operator. It is defined as the `TemporalAggregator` in line 14. Its parameters are defined in line 15. The aggregation function is set as `Mean` (line 16) and the aggregation interval is set to six months (line 17). The operator also has one input operator that is defined in line 23: `GDALSource`. It only has one parameter in line 25 that defines the input time series it uses: `temp_month`. It depicts the monthly average land surface temperature for each month of 2001 and is provided by NASA Earth Observations (NEO) [11].
The query aggregates the complete input time series into two output rasters each spanning six months. It aggregates the rasters by calculation the mean of each cell in the aggregation time interval. Figure 16 shows the first raster of the input time series and Figure 17 shows the first of two aggregated output rasters. They are not colored because IPRTS does not handle colorization of raster data at the moment. To display these rasters, their `GeoTiff` files were transformed into `png` files with the `gdal_translate` utility. In the rasters, light pixels depict high and dark pixels depict low temperature. It can be seen that the northern parts of the world are warmer in the aggregated output than in the input January raster because the aggregated raster depicts the mean temperature between January and June. Parts at the northern and southern poles of the map are completely white because they contain the nodata value.

[11]`https://neo.sci.gsfc.nasa.gov/view.php?datasetId=MOD_LSTD_CLIM_M`

Code 7: First part of the example query. It defines the query rectangle.

```
{
  "query_rectangle" : {
    "resolution" : {
      "x" : 3600, "y" : 1800
    },
    "temporal_reference" : {
      "type" : "UNIX", "start" : 978307200, "end": 1007251200
    },
    "spatial_reference" : {
      "projection": "EPSG:4326",
      "x1": -180, "x2": 180, "y1": -90, "y2": 90
    },
    "order" : "Temporal",
    "tileRes" : {
      "x" : 1000, "y" : 1000
    }
  },
  "operator" : ...
}
```

Code 8: Second part of the example query. It defines the operator tree.

```
{
  "query_rectangle" : { ... },
  "operator" : "geotiff_export",
  "params" : {
    "time_format" : "%Y-%m-%dT%H:%M:%S",
    "filename" : "example_query_%%%TIME_STRING%%%.tiff"
  },
  "sources" : [
    {
      "operator" : "order_changer",
      "params" : { },
      "sources" : [
        {
          "operator" : "aggregator",
          "params" : {
            "function" : "Mean",
            "time_interval" : {
              "unit" : "Month", "length" : 6
            }
          },
          "sources" : [
            {
              "operator" : "gdal_source",
              "params" : {
                "dataset" : "temp_month"
              },
              "sources" : [ ]
            }
          ]
        }
      ]
    }
  ]
}
```

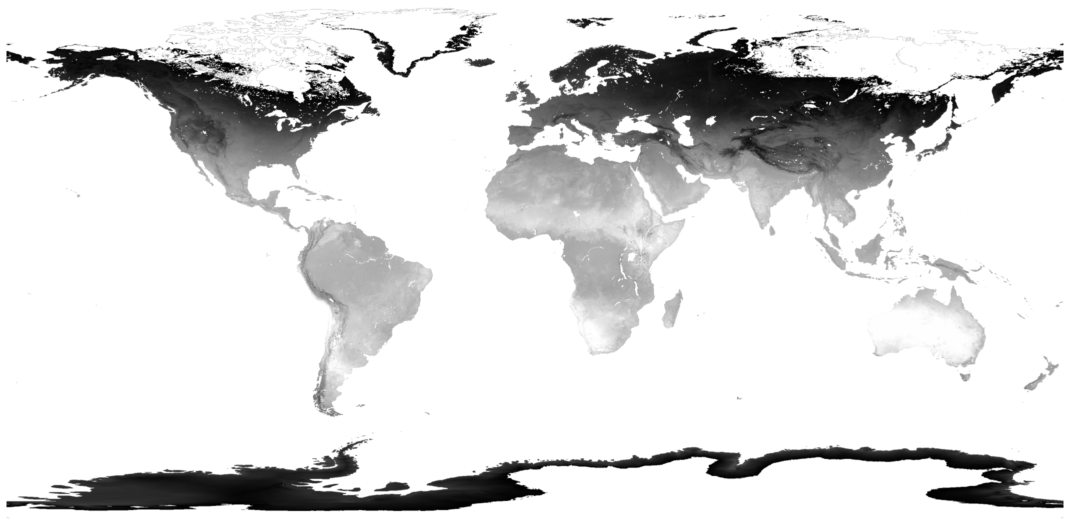


Figure 16: First input raster of the source time series, valid from 01/01/2001 to 01/02/2001.

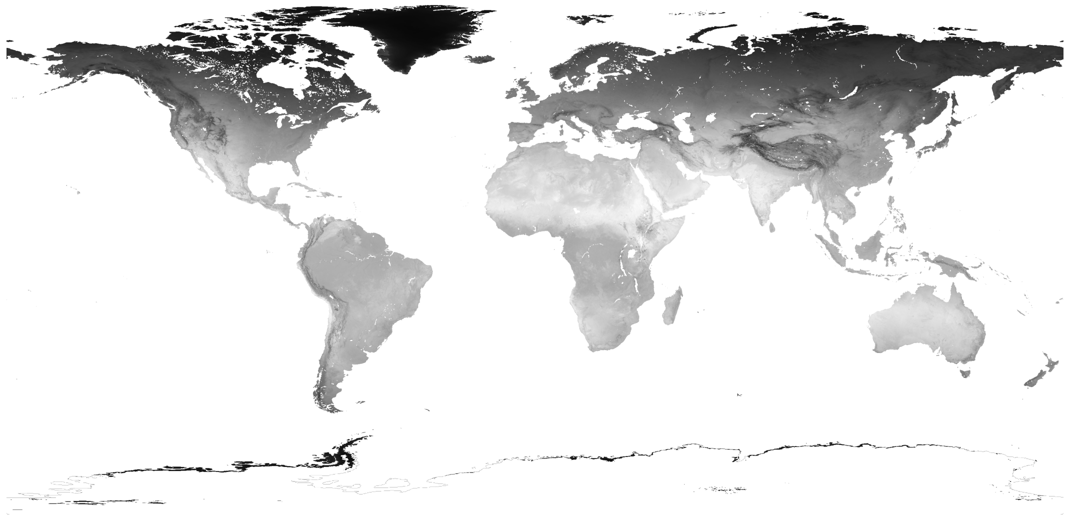


Figure 17: First aggregated raster of the output time series, valid from 01/01/2001 to 01/07/2001.

8 Performance Benchmarks

In this section, different queries to IPRTS are used as benchmarks to explore the performance of the system. To make the queries reproducible, the name of each executed query file will be provided. The query files can be found on the source code repository [12].
For benchmarking the executable `rts_benchmark_query` was implemented in IPRTS. It opens a JSON query file, creates an operator tree from it, and executes it. It repeatedly measures the execution time of the query and saves it to a file. The executable requires three program arguments: the location of the query file, name of the output text file, and the number of repeated executions. It measures not only the total query execution time but also the time spent in the source and the consuming operator. The times are measured in milliseconds with the `high_resolution_clock` from the `std::chrono` library. To compare certain features, different versions of the executable will be used for different benchmarks. The branch and commit that were used for building the executable will be named for each query. All executables are built in release mode with optimization flag `O3`.
The computer used to execute all benchmarks has an Intel i5-4570 CPU (four cores at 3.20GHz) with 8 GB of main memory and was running Ubuntu 18.04.1 LTS. Files were loaded from and written to an SSD drive. The executables were executed from the command line in the `target/bin` directory of the source code repository.

Two different datasets are used in these benchmarks. The first contains satellite data from *Meteosat Second Generation* and will be called *meteosat* in the following [13]. The dataset contains rasters covering Europe. As GeoTiff files can save multiple layers of data in one raster, each raster of the time series provides multiple layers containing different measurements like surface temperature, or reflectance. Layer one that contains reflectance data will be used in the benchmark. The available time series spans from 1/1/2011 at 00:00:00 to 1/6/2011 at 14:45:00, containing over 500 rasters in total. Each raster has a resolution of 767×510 pixel.
The second dataset is provided by NASA Earth Observations (NEO) [14] and was already used in Section 7. It covers the average land surface temperature at day in monthly rasters. It will be referred to as *NEO* time series. The rasters cover the whole world and have a resolution of 3600×1800 pixel. The time series contains twelve rasters and spans from January 2001 to December 2001.

In Section 8.1 the performance difference between using and not using random tile access in different operators will be explored. Section 8.2 benchmarks the execution times for two queries

[12]`https://github.com/SoerenHoffstedt/raster-time-series/tree/master/test/query`
[13]`https://www.eumetsat.int`
[14]`https://neo.sci.gsfc.nasa.gov/view.php?datasetId=MOD_LSTD_CLIM_M`

that only differ in their tile order. Section 8.3 tests the usage of sampling a time series and compares the results of the sampled with the not sampled time series. Section 8.4 explores how the raster value extraction operator performs for different point distributions. Finally, Section 8.5 compares the performance of queries in IPRTS with similar queries to rasdaman.

8.1 Random Tile Access Comparison

While different concepts for implementing focal operators were discussed in Section 5.5, only the fourth concept was implemented. It adds random tile access to IPRTS (see Section 6.9). To compare the concepts, the operators that would make use of random tile access were also implemented without random tile access (concept one). These operators are `Convolution` and `OrderChanger`.

For each benchmark, the query is executed with two executables. The first is the implementation of concept one, is based on the `NoRandomAccessConvolutionAndOrderChanger` branch [15], and does not use random tile access. The second executable implements the random tile access of concept four and is based on the `master` branch [16]. Sections 6.10.7 and 6.10.8 explained the implementation of the `Convolution` and `OrderChanger` with and without usage of random tile access.

Three benchmarks were executed to test the random tile access. The first tests the performance impact for the `OrderChanger` between both implementations. The second tests how the random tile access implementation of the `TemporalAggregation` operator impacts the performance. The last query tests the main use case of random tile access: the `Convolution` operator. In Section 8.1.1 the results will be analyzed to compare the strength and weaknesses of the concepts.

Benchmark I - Order Changing

The first benchmark explores the performance differences of the `OrderChanger` operator for concept one and four. In the query, the operator changes the tile order from spatial to temporal. It consists of four operators: `GeoTiffExporter`, `OrderChanger`, `ExpressionOperator`, and `GDALSource`. The `ExpressionOperator` has no performance difference between iterating and random tile access. As the tile order is changed from spatial to temporal, the rasters are exported in temporal order and loaded in spatial order. The benchmark is executed for two different queries that use the *meteosat* time series as input and process a different number of total rasters. By executing these two queries, it is explored if the `OrderChanger` performs differently based on the number of rasters processed.

The first query is called `benchmark_random_access_1_144.json` and creates 144 output

[15]Hash of the commit used for concept one: 6c6d69e6a477a9e5388f894c27b088f771aecf47

[16]Hash of the commit used for concept four: 6aee214f0bca5805af7dfb28510c94a8d1f35317

rasters. The second query is called `benchmark_random_access_1_528.json` and creates 528 output rasters. Both queries use a tile size of 256 × 256 pixel. The difference count of output rasters is caused by the later end time of the second query.

	Concept One	Concept four
Total Query	2856.32	2883.64
Consuming Operator	1276.84	1288.05
Source Operator	822.99	827.08

Table 1: Mean execution times of benchmark I in ms (144 rasters).

	Concept One	Concept four
Total Query	26971.27	27292.70
Consuming Operator	4806.53	4944.77
Source Operator	19167.00	19484.10

Table 2: Mean execution times of benchmark I in ms (528 rasters).

Tables 1 and 2 show the mean execution times of both queries with the executables from both concepts. The first query was executed 100 times. The second query was executed only 30 times because of its longer execution time.
For the first query, concept four is 0.96% slower, and for the second query, it is 1.19% slower. The difference might be caused by a slight overhead of the `getDescriptor` method that has to execute some additional calculations, for example, for the spatial domain of a tile. The difference in execution time is spread equally over consuming, source, and other operators. The benchmark also shows that loading and storing all tile descriptors of a time series at once, as done by the `OrderChanger` that does not use random tile access, does not have a negative performance impact.

Benchmark II - Temporal Aggregation
The second benchmark analyzes the performance impact of the random tile access implementation of the `TemporalAggregator`. Because the operator has to access multiple successive rasters, the operator has to re-instantiate its input operator in the `getDescriptor` and perform a sub-query to support the random tile access of concept four. The query `benchmark_random_access_2.json` swaps the expression operator from the last benchmark with the temporal aggregation operator. Additionally, the query uses the *NEO* time series because it explores the impact of the aggregation operator and a high number of rasters is not expected to make a difference for this use case because the number of tiles that have to be accessed for a single output would not be affected by the total length of the time series. The

query builds on top of the result of the first benchmark that showed that the random tile access does not make a significant difference for the `OrderChanger`.

	Concept One	Concept four
Total Query	448.16	657.99
Consuming Operator	154.93	161.72
Source Operator	103.22	210.74

Table 3: Mean execution times of benchmark II in ms.

Table 3 shows that the average execution time of concept four is 209.83ms and thus nearly 50% slower than concept one. Because benchmark I shows that the `OrderChanger` has a minimal performance difference between both concepts, this difference is caused by the `TemporalAggregator`. The increase in execution time is split between the source operator and the processing operators. The consuming operator is only slightly slower.

Benchmark III - Convolution

The third benchmark compares the performance of the convolution operator between concept one and four. The query `benchmark_random_access_3.json` only contains `Convolution` as a processing operator and uses the *NEO* time series as input. To explore the improvements of caching the tiles before the convolution operator, a second query `benchmark_random_access_3_cache.json` is executed. It is the same query, but adds the `RasterCache` before the `Convolution` operator.

	Concept One	Concept four
Total Query	1571.95	1510.33
Consuming Operator	429.47	411.59
Source Operator	265.06	260.65

Table 4: Mean execution times of benchmark III (no caching) in ms.

	Concept One	Concept four
Total Query	1334.75	1278.83
Consuming Operator	402.25	401.95
Source Operator	104.47	104.45

Table 5: Mean execution times of benchmark III (with caching) in ms.

Tables 4 and 5 both show that concept four is faster than concept one. With caching, concept one is 4.4% slower than concept four, and without caching, it is 4.1% slower. To compare the query processing with and without caching, concept four is 18.1% and concept one 17.8% faster with caching.

8.1.1 Result discussion

The benchmarks show mixed results for both concepts. The main motivation of concept four was to handle focal operations better and potentially faster. The implementation of the `Convolution` operator became cleaner by adding the random tile access of concept four. It also shows small performance improvements, and thus concept four reaches its initial goal.
Another important advantage that concept four was supposed to provide was its potential improvements to the `OrderChanger`. When changing the order from spatial to temporal, it does not have to load and save all input descriptors in the beginning. Instead, it stores only the first tile of a raster and loads the other tiles with the random tile access. Even though benchmark I shows that this is slightly slower than the concept one, the implementation of the operator is a bit easier.
In the normal iterator-based processing of tiles, only `Convolution` and `OrderChanger` of the implemented operators make use of the random tile access. But all the other operators have to implement the `getDescriptor` method and also make sure that they are not changing the state of their input operators. The implementation and performance penalty of the `TemporalAggregator` show these negative effects. Also the problems the random tile access adds to the `Sampler` must be taken into account when evaluating the concepts.
To conclude, these results indicate that the advantages of random tile access are too small to justify the complexity of its implementation, and the restrictions it adds to some operator implementations. Further assessment of the second concept should be interesting, especially when its interaction with a system-wide caching mechanism can be explored. The second concept splits the tile size into a window and a step size. This would make it possible that focal operators do not need to access multiple tiles for one output tile.

8.2 Temporal and Spatial Order

When processing a raster time series in spatial order, the `GeoTiffExporter` as well as the `GDALSource` operator have to keep multiple data structures from the GDAL library open to read from and write into files on disk. Thus, it is assumable that processing a raster time series in temporal order has advantages in performance and memory usage. This assumption is explored by benchmarking four different queries.
Two queries access the *NEO* time series and the other two access the *meteosat* time series. Queries on the same input time series have the same query except that one processes the tiles

in spatial and the other in temporal order. The *NEO* queries process twelve rasters [17] and the *meteosat* queries process 240 rasters [18]. The executable used for this benchmark is based on the `master` branch [19].
Table 6 and the bar graph of Figure 18 show that the query that processes twelve rasters executes equally fast for both tile orders. But the queries processing 240 rasters show a huge difference. In temporal tile order, the execution takes less than half of the time of the query in spatial order.

	Spatial Order	Temoral Order
Total Query (12 rasters)	1232.37	1284.60
Total Query (240 rasters)	12562.00	5369.40

Table 6: Mean execution times of spatial vs. temporal order benchmark in ms .

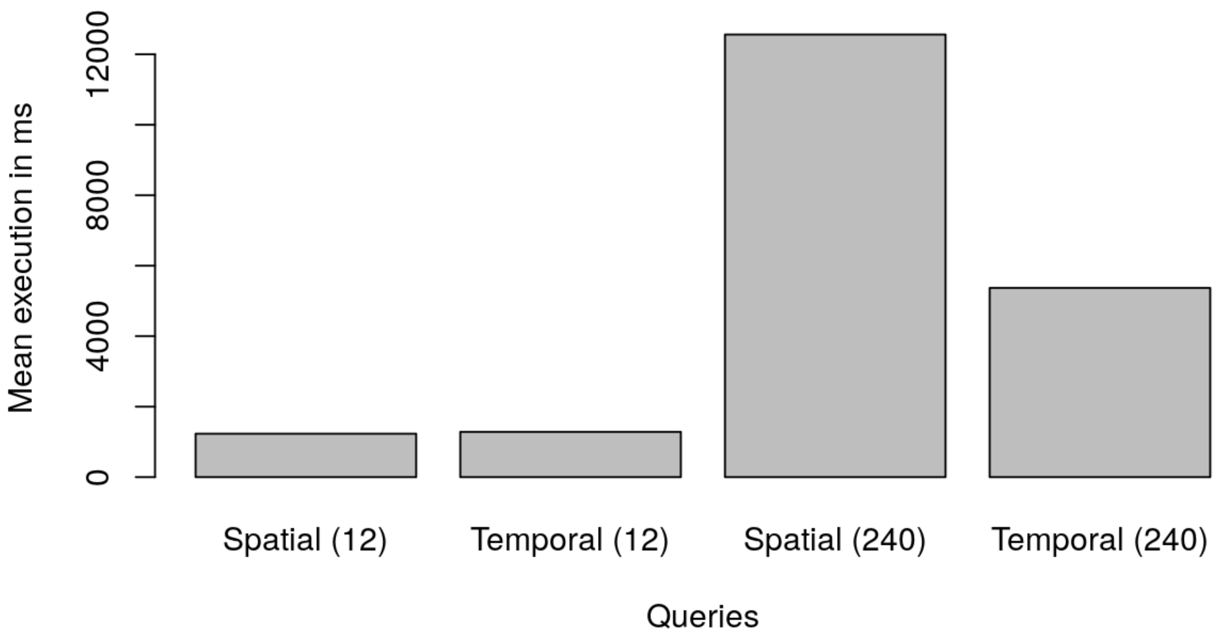


Figure 18: Bar graph of mean execution of spatial vs. temporal benchmarks in ms.

[17]Query files: `benchmark_spatial.json` and `benchmark_temporal.json`.
[18]Query files: `benchmark_spatial_big.json` and `benchmark_temporal_big.json`.
[19]The hash of the used commit is: `6aee214f0bca5805af7dfb28510c94a8d1f35317`

8.3 Sampling of a Time Series

As described in the problem analysis (see Section 3), sampling a time series is an important use case for IPRTS. This benchmark compares the execution time of two queries. The first query is executed normally and the second query contains a sampling operator that skips every second raster of the time series. The queries aggregate rasters from the *meteosat* time series in six-hour intervals. The total query is executed over a time span of one and a half days, resulting in six aggregated output rasters. The aggregation calculates the local max value for each cell.

The first query `benchmark_sampling_no.json` contains three operators: `GDALSource`, `TemporalAggregator`, and `GeoTiffExporter`. The second query `benchmark_sampling.json` adds a `Sampler` that skips every second raster before the aggregation operator. The queries were each executed 10 times for this benchmark on the same executable [20].

	Sampling	No Sampling
Total Query	884.40	3758.10
Consuming Operator	10.40	14.40
Source Operator	451.30	2943.70

Table 7: Mean execution times of sampling benchmark in ms.

Table 7 and Figure 19 show that the sampled query is executed much faster. An interesting aspect is that the execution of the unsampled query is not doubling but quadrupling the execution time in comparison to the sampled query. The performance bottleneck is the source operator. The sampled query spends 433.1 ms and the unsampled query spends 814.4 ms outside of the source operator. This time roughly doubled for the unsampled query as it would be expected because it is processing twice the amount of rasters. The increase of time is spent in the source operator.

A third query was executed to analyze the difference in quality between both queries. `benchmark_sampling_diff.json` executes both queries from above and uses them as input for an expression operator that calculates the absolute difference between both time series.

The difference between both query results is analyzed by calculating the average, minimum, and maximum cell value for every output raster individually. The analysis uses the `RasterAnalyzer` operator that was implemented for this use case. It reads all cell values and writes the mentioned statistics into a file. The *metosat* raster time series has `uint16` as the data type. The maximum cell values found in the unsampled raster time series is 22307.

Table 8 shows the average cell values of each of the six output rasters for the sampled, unsampled, and difference time series. It is important to note that each raster of the resulting

[20] Executable was build from the commit `207e743c9802fb8a322f3ac70a74ed1f4acb50da` on branch `ExpressionCalculatingAbsoluteValues`.

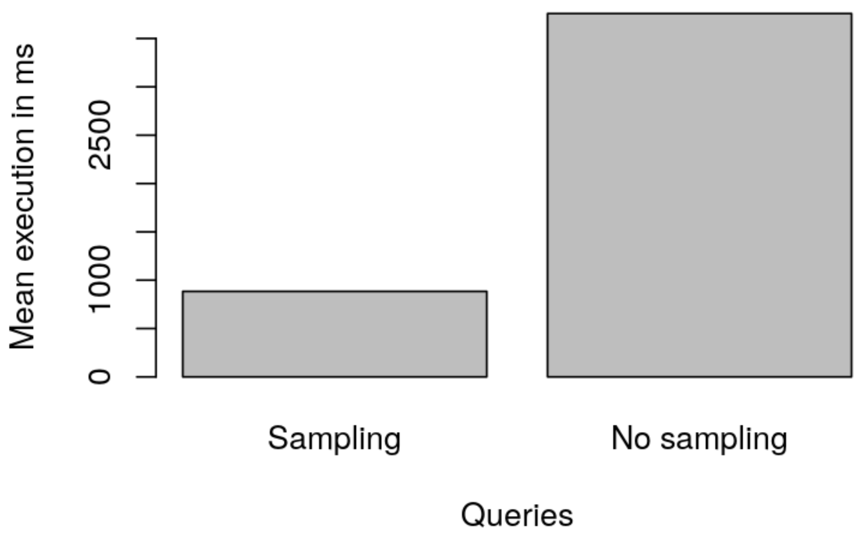


Figure 19: Boxplot of sampling benchmark results.

time series has vastly different average cell values. This is because the input raster time series measure reflectance which is very low at night. The difference in average values between the raster is based on the time period they represent.

Raster	Difference	Sampling	No Sampling
0	11.77	13.42	4.17
1	165.26	5797.14	5876.53
2	350.00	5039.35	5389.35
3	1.75	1.67	3.02
4	4.67	3.98	4.24
5	184.49	5567.88	5752.36

Table 8: Average cell values of rasters for each query.

The highest difference between the sampled and unsampled queries can be found for raster two. The difference of 350 is 6.49% of the average of the unsampled time series and 1.57% of the total maximum value of the time series. The average cell difference in raster one is 2.81% of the unsampled time series and for raster five it is 3.21%. For raster zero, three, and four the total values are so small that the proportion to the unsampled average is not relevant.
These differences in the results are small enough that the sampled query provides a good

approximation of the actual result. Especially, when the processed time series gets bigger, the difference in execution time will be important for exploratory research.

8.4 Raster Value Extraction

In the problem analysis (see Section 3), it was also outlined that the raster value extraction operator can profit from lazy loading of raster data when not every tile of a raster time series has to be accessed by the operator. In Section 6.7 it was explained how the implementation of IPRTS allows skipping rasters and tiles of a time series. The performance advantages of this design for raster value extraction are explored in this benchmark.
Based on the distribution of the requested points, the raster value extraction operator has to load a different amount of tiles from a raster time series. To show the performance impact of loading less tile data, this benchmark executes four different queries. They process a time series consisting of twelve rasters from the *NEO* time series with a resolution of 3600 × 1800 pixels. To make different point distributions easy, the tile size is 1800×1800 pixels so that every raster is split only in two tiles. The executable is build from the `master` branch [21] and each query was benchmarked 100 times. The queries only consist of `GDALSource` and `RasterValueExtraction` operators. Each query requests the pixel values of 24 points but the point distribution differs.

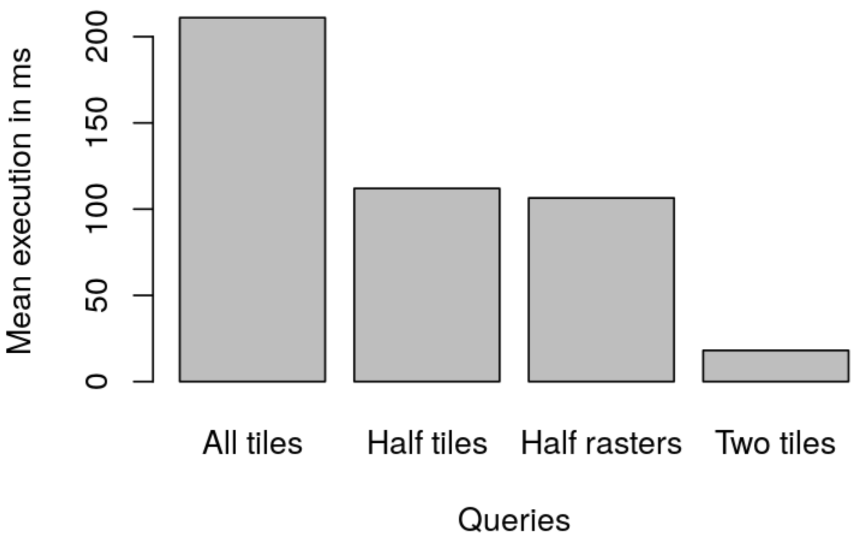


Figure 20: Boxplot of raster value extraction benchmark results.

[21]The hash of the used commit is: `6aee214f0bca5805af7dfb28510c94a8d1f35317`

	All tiles	Half tiles	Half rasters	Two tiles
Total Query	211.00	112.00	106.46	18.07
Consuming Operator	0.15	1.57	0.11	0.00
Source Operator	193.77	99.87	97.40	16.71

Table 9: Mean execution times of raster value extraction benchmarks in ms.

The *all tiles* query requests one point for each tile of the time series, *half tiles* requests two points for one tile of each raster, *half rasters* requests two points on each tile of half of the rasters of the time series, and *two tiles* requests twelve points on two tiles of two different rasters [22].
The execution time of the four queries scales roughly with the number of loaded tile data as seen in Table 9 and Figure 20. This is an expected result and confirms the performance advantages of lazy loading of tile data for raster value extraction. Notable is the difference between the *half tiles* and *half rasters* queries. *Half tiles* is 5.2% slower then *half rasters* and it is caused by the different amount of rasters that are opened by both queries.

8.5 Comparison to Rasdaman

This benchmark compares the performance of IPRTS with rasdaman. Two different test cases were executed in both systems, and both use rasters from the *NEO* time series. The rasters were imported into a rasdaman database using the `wcst_import.sh` tool under the name *test_month*. Queries to rasdaman were executed with the `rasql` executable from the official rasdaman debian packages. Both communicate with a local database that is installed by the package. The execution time for the rasdaman queries were timed with the linux command `time`. The IPRTS queries were benchmarked 100 and the rasdman queries 30 times. The queries for IPRTS were executed with an executable from the `master` branch [23].
The first test exports the first raster from the time series as a GeoTiff to disk. For IPRTS the `benchmark_rasdaman.json` query was used. In rasdman the query was executed with the following command: `rasql -q "select encode(m[0,0:1800,0:900], \"tiff\") from test_month as m" -out file`.
The second test aggregates all twelve rasters of the time series into one output raster by adding all tiles together. For IPRTS the `benchmark_rasdaman_aggregation.json` query was used. The query uses the `TemporalAggregation` operator. In rasdaman the query was executed with the following rasql query: `rasql -q "select encode( condense + over x in [-11:0] using m[x[0], *:*, *:*] , \"tiff\" ) from test_month as m" -out file`.

[22]The query files are: `benchmark_extraction_all_tiles.json`, `benchmark_extraction_half_tiles.json`, `benchmark_extraction_half_rasters.json`, and `benchmark_extraction_two_tiles.json`

[23]The hash of the used commit is: `6aee214f0bca5805af7dfb28510c94a8d1f35317`

The results in Table 10 and Figure 21 show that IPRTS is significantly faster for both queries. The first test is about five times and the second about 30 times faster in IPRTS. The conclusion should not be that IPRTS is 30-times faster than rasdaman. Rasdaman is a more complex system, that handles optimizations and the overhead of a complete database management system. Due to limited familiarity with rasdaman, it is questionable if the rasdaman queries are the optimal way to handle. However, this comparison shows that IPRTS is performing well and in the dimensions of other raster time series processing systems. It is also interesting that the aggregation of twelve rasters is 30 times slower while exporting one raster is only 5 times slower. This shows that the difference in speed cannot be solely based on the overhead of the `rasql` executable.

	IPRTS (One)	Rasdaman (One)	IPRTS (Agg.)	Rasdaman (Agg.)
Total Query	31.20	164.53	936.31	29441.10

Table 10: Mean execution times of rasdaman comparison benchmarks.

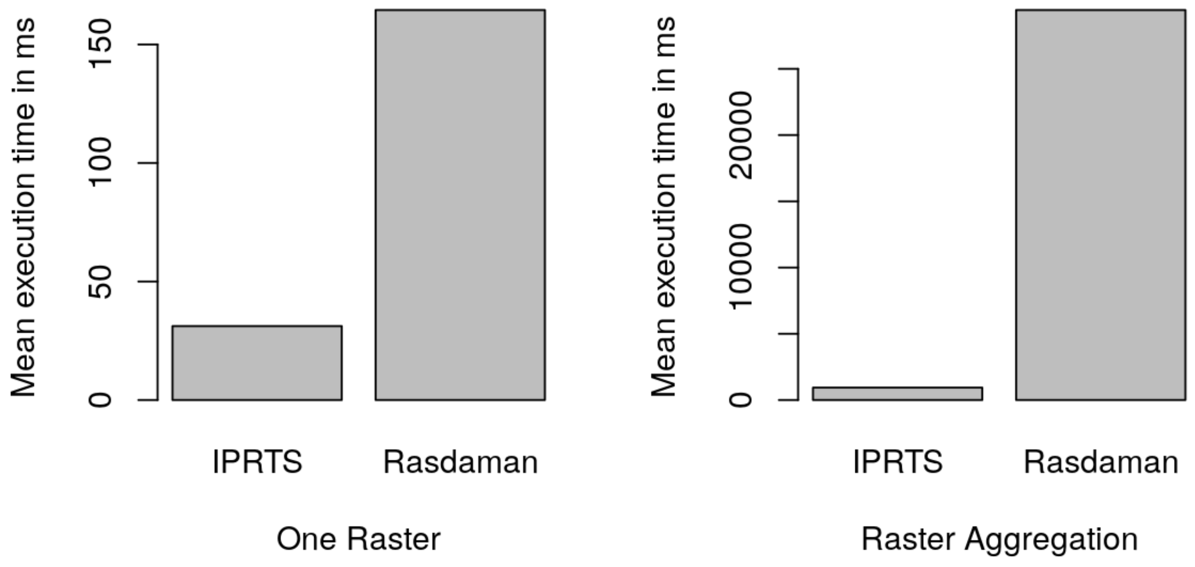


Figure 21: Mean execution times for both queries with IPRTS and rasdaman.

9 Conclusion

This Bachelor thesis presented *Iterative-Processing of Raster Time Series*, a concept and prototypical implementation for processing raster time series that was developed with the goal to be integrated into MAPPING. First, this thesis presented basic concepts about raster time series in Section 2, analyzed the problems of processing raster time series in Section 3, and introduced other software that processes raster time series in Section 4. Because the size of single rasters and the number rasters in a time series can be huge, raster time series must be processed by *iterating* over raster *tiles* individually. To avoid loading of raster data that is not actually needed, operators have to processes *descriptors* of tiles that describe them with metadata and provide a function for loading the data. As explained in Section 5, IPRTS uses the concepts of tiles, descriptors, and iteration. It also allows for two different orders of processing tile descriptors. Section 6 presented the implementation of a prototype of IPRTS, Section 7 showed an example query to IPRTS, and Section 8.1 tested the performance of the prototype in different benchmarks.

The goal was that IPRTS is a flexible concept for processing raster time series. The provided implementation shows that it supports local and focal operations with the expression and convolution operator. It also allows changing the temporal resolution of a raster time series with the temporal aggregation and temporal overlap operators. The raster value extraction operator shows that IPRTS can combine raster time series with vector data.
As the concept utilizes lazy loading of raster data by using the *descriptors* and *iterator-based* operators, IPRTS can flexibly process raster time series and only has to access the raster data that is actually needed to calculate a result.
In Section 3 two scenarios were outlined to show the advantages of these concepts and the benchmarks in Section 8 confirmed both. The first was the raster value extraction operator that reads pixel values from rasters for a list of spatial-temporal points. Based on the distribution of these points in the raster time, not all raster tiles must be loaded into IPRTS. The benchmark in Section 8.4 shows that the execution time of a raster value extraction query scales linearly based on the number of tiles that have to be loaded. The second scenario is sampling a raster time series to reach faster results in exploratory research. By sampling rasters of a time series with high temporal resolution, the processing time can be reduced while the result is still a good approximation of the unsampled result. This was demonstrated successfully with a benchmark in Section 8.3. Both scenarios demonstrate that IPRTS reaches the goal of providing flexible raster time series processing.

The benchmark in Section 8.2 explores the performance differences of processing a raster time series in spatial and temporal tile order. It shows that spatial tile ordering is significantly slower for a large raster time series. The same result can be seen in the sampling benchmark in Section 8.3 where sampling halves the number of processed rasters. Because the benchmark aggregates rasters, it is using spatially ordered tiles. The execution time for processing half of the rasters is a fourth of the processing time of the complete raster time series. The sampling benchmark also shows that the performance difference is based on the `GDALSource` operator. Consequently, its implementation of spatial ordering has to be analyzed further. In temporal tile order, every input raster must only be opened once and every tile can be read from it successively. In spatial order, the operator keeps open the GDAL data structures to prevent opening them multiple times. If the performance difference is inherent to not reading tiles successively from files, the spatial tile ordering has to be reconsidered. A strategy to limit the number of GDAL data structures that have to be kept open is splitting a query temporally into multiple sub-queries.

The benchmarks in Section 8.1 explore the performance differences between two concepts of handling focal operations in IPRTS. Because of the iterator-based design, an operator usually accesses only one tile at a time. In the base concept of IPRTS, it the convolution operator has to use a workaround where it loads all descriptors of a tile and saves them in a list. To create its output descriptors, the operator then reads the needed descriptors from the list. As another solution, random tile access was implemented in IPRTS. It allows operators to load any tile of the current raster at request. As it was concluded in Section 8.1.1, using random tile access provides no significant performance advantages. On the contrary, the random tile access implementation for the `TemporalAggregation` operator has severe performance downsides because it has to re-instantiate its input operator to access tiles from multiple rasters of the time series. Random tile access makes the convolution operator more structured but also adds restrictions that each operator has to satisfy, like having no temporal gaps in a raster time series. Therefore, it can be concluded that random tile access is not a valuable addition to IPRTS.

For future development of IPRTS, it must be explored how the concept can be integrated into MAPPING. At the moment, IPRTS is missing key elements required for VAT, like tracking of provenance information, a mechanism for result caching, or colorizing information. For the raster value extraction operator, integration with the vector data concept of MAPPING is needed. In this context, an iterator-based approach for processing vector data can also be explored. IPRTS itself can be improved by query rewriting or by choosing the processing order of an operator tree algorithmically based on the used operators. Most operators can also be extended to provide more processing options, like the expression operator.

References

[1] P A. Burrough and Rachael McDonnell. Principle of geographic information systems. 1998.

[2] Christian Beilschmidt, Johannes Drönner, Michael Mattig, Marco Schmidt, Christian Authmann, Aidin Niamir, Thomas Hickler, and Bernhard Seeger. Vat: A scientific toolbox for interactive geodata exploration. *Datenbank-Spektrum*, 17(3):233–243, Nov 2017.

[3] C.D. Tomlin. *Geographic Information Systems and Cartographic Modeling*. Prentice Hall series in geographic information science. Prentice Hall, 1990.

[4] G.X Ritter, J.N Wilson, and J.L Davidson. Image algebra: An overview. *Computer Vision, Graphics, and Image Processing*, 49(3):297 – 331, 1990.

[5] A. G. Gutierrez and P. Baumann. Modeling fundamental geo-raster operations with array algebra. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, pages 607–612, Oct 2007.

[6] Peter Baumann, Dimitar Misev, Vlad Merticariu, Bang Pham Huu, and Brennan Bell. rasdaman: Spatio-temporal datacubes on steroids. In *SIGSPATIAL/GIS*, pages 604–607, 2018.

[7] Piero Campalani, Xinghua Guo, and Peter Baumann. Spatio-temporal big data – the rasdaman approach, 2014.

[8] Ralf Güting, Thomas Behr, and Christian Düntgen. Secondo: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 33:56–63, 2010.

[9] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. Distributed secondo: an extensible and scalable database management system. *Distributed and Parallel Databases*, 35(3):197–248, Dec 2017.

[10] Dirk Zacher. Secondo user manual: Working with raster data. `http://dna.fernuni-hagen.de/secondo/files/Documentation/Algebras/Tile/Working_With_Raster_Data.pdf`, 2014. Accessed: 02/02/2019.

[11] n.n. Examples for using GISAlgebra. `http://dna.fernuni-hagen.de/secondo/files/Documentation/Algebras/Tile/Examples_for_GISAlgebra.pdf`. Accessed: 10/02/2019.

[12] Geotrellis docs. `https://docs.geotrellis.io/`. Accessed: 01/02/2019.

[13] C++ reference: Lambda expressions (since c++11). `https://en.cppreference.com/w/cpp/language/lambda`. Accessed: 17/01/2019.

[14] C++ reference: Iterator library. `https://en.cppreference.com/w/cpp/iterator`. Accessed: 10/02/2019.

[15] C++ reference: Range-based for loop (since c++11). `https://en.cppreference.com/w/cpp/language/range-for`. Accessed: 10/02/2019.

[16] Arcgis for desktop: Convolution function. `http://desktop.arcgis.com/en/arcmap/10.3/manage-data/raster-and-images/convolution-function.htm`. Accessed: 04/02/2019.