FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Hierarchical Semi-Sparse Cubes - scalable solution for combining dimensionally multi-modal big data

by

Ing. Jiří Nádvorník

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Doctoral study programme: Informatics
Department of Theoretical Computer Science

Prague, August 2023

Supervisor:
prof. Ing. Pavel Tvrdík, CSc.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Co-Supervisor:
RNDr. Petr Škoda, CSc.
Astronomical Institute of the Czech Academy of Sciences
Fričova 298
251 65 Ondřejov
Czech Republic

Abstract and contributions

Since Moore's law applies also to data detectors, the volume of data collected in astronomy doubles approximately every year. A prime example is the upcoming Square Kilometer Array (SKA) survey that will produce approximately 8.5 exabytes over the first 15 years of service, starting in the year 2027. Storage capacities for these data have grown as well, and primary analytical tools have also kept up. However, the tools for combining big data from several such detectors still lag behind. Finding interesting information in the Big data is relatively easy, but inferring new knowledge based on this information is possible only when it is put into context. That requires to combine the information from multiple data sources.

In this thesis, we present the Hierarchical Semi-Sparse Cube (HiSS-Cube) framework. It aims to provide highly parallel processing of combined dimensionally multi-modal big data.

We tested the scalability and efficiency of HiSS-Cube on big astronomical spectroscopic and photometric data obtained from the Sloan Digital Sky Survey (SDSS). The performance of HiSS-Cube is bounded by the I/O bandwidth and I/O operations per second of the underlying parallel file system, and it scales linearly with the number of I/O nodes.

In particular, the main contributions of the dissertation thesis are as follows:

- We have developed the concept of semi-sparse data as a combination of dimensionally multi-modal data.
- We designed and developed the HiSS-Cube framework that supports the combination of semi-sparse data up to the PB-scale efficiently.
- We created a HiSS-Cube File structure based on HDF5 that scales with the number of I/O nodes of the underlying parallel file system.
- We adapted the HiSS-Cube framework to support additional preprocessing strategies implemented as Python modules. Therefore, adding support for other types of dimensionally multi-modal data from astronomy, bioinformatics, chemistry, physics,

and Earth sciences, is performed by implementing another Python module within the HiSS-Cube framework.

- We tested the scalability of the HiSS-Cube framework to hundreds of TB of data from the SDSS. The whole SDSS survey can be processed in approximately one hour.
- We performed an extensive analysis of the performance of the HiSS-Cube framework with semi-sparse data on the `Lustre` parallel file system.
- We developed a standardized metadata annotation that enables seamless interactive visualization of pairs of spectra and image regions relevant to those spectra.

Keywords:
big data, multi-modal data, multi-dimensional data, semi-sparse data, HDF5, parallel I/O, astroinformatics, Lustre file system

Abstrakt

Vzhledem k tomu, že Moorův zákon se vztahuje i na detektory, které data produkují, objem pozorovaných dat v astronomii se každý rok zhruba zdvojnásobuje. Důkazem je například nová "Square Kilometer Array (SKA)" astronomická přehlídka oblohy, která vyprodukuje zhruba 8.5 exabytů dat ve svých prvních 15 letech provozu, který začíná v roce 2027. Kapacita datových úložišť roste stejně rychle, stejně jako schopnost zpracovávat tyto data primárními analytickými nástroji. Kde přichází problém je schopnost kombinovat velká data z několika takovýchto detektorů. Vytěžit informace z těchto velkých dat je relativně snadné, ale odvozování nových znalostí na základě těchto informací je možné jen pokud se na ně díváme v kontextu. To vyžaduje kombinaci dat z několika zdrojů. V této dizertaci představujeme softwarové řešení „Hierarchical Semi-Sparse Cube“ (HiSS-Cube). Klade si za cíl masivně paralelní kombinaci a zpracování dimenzionálně multi-modálních velkých dat. Otestovali jsme škálovatelnost a efektivitu na astronomických velkých datech ze spektrometrických a fotometrických přehlídek "Sloan Digital Sky Survey" (SDSS). Ověřili jsme, že HiSS-Cube je omezen I/O propustností a I/O operacemi za sekundu paralelního file systému, na kterém software běží, tedy že škáluje lineárně s počtem I/O uzlů. Hlavní přínosy

této dizertace jsou:

- Vyvinuli jsme koncept "Semi-Sparse" dat jako kombinaci dimenzionálně multi-modálních dat.
- Navrhli a vyvinuli jsme HiSS-Cube framework, který podporuje efektivní kombinaci Semi-Sparse dat pro petabyty.
- Vytvořili jsme strukturu HiSS-Cube framework souboru založenou na HDF5, která škáluje s počtem I/O uzlů paralelního file systému, na kterém software běží.
- Adaptovali jsme HiSS-Cube framework tak, aby podporoval jiné preprocessing strategie implementované jako Python moduly. Díky tomu stačí přidat nový Python modul pro podporu jiného typu multi-modálních dat z astronomie, bioinformatiky, chemie, fyziky a vědách o Zemi.

- Otestovali jsme škálovatelnost HiSS-Cube framework pro stovky TB dat z SDSS přehlídky. Zpracování celé SDSS přehlídky trvá zhruba jednu hodinu.
- Extenzivně jsme analyzovali výkon HiSS-Cube frameworku na Lustre paralelním file systému.
- Vyvinuli jsme standardní anotaci metadat, která umožňuje jednoduchou interaktivní vizualizaci párů spekter a oblastí v obrázcích, která překrývají tato spektra.

Klíčová slova:
big data, multi-modální data, vícerozměrná data, semi-sparse data, HDF5, paralelní I/O, astroinformatika, Lustre file systém

Acknowledgements

We thank Kai Polsterer for the initial ideas and later consultation of our approach and also Markus Demleitner for his comments. We also thank Mark Taylor for a very swift fix when we discovered a bug in TOPCAT.

We particularly thank to Gerd Heber who invested a lot of time and helped us to understand the key strengths of HDF5 and how to exploit them.

The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01-/0.0/0.0/16_019/0000765 "Research Center for Informatics" and e-INFRA CZ (ID:90140). Astronomical Institute of the Czech Academy of Sciences is supported by the project RVO:-67985815.

Funding for the Sloan Digital Sky Survey IV has been provided by the Alfred P. Sloan Foundation, the U.S. Department of Energy Office of Science, and the Participating Institutions. SDSS-IV acknowledges support and resources from the Center for High-Performance Computing at the University of Utah. The SDSS web site is www.sdss.org.

SDSS-IV is managed by the Astrophysical Research Consortium for the Participating Institutions of the SDSS Collaboration including the Brazilian Participation group, the Carnegie Institution for Science, Carnegie Mellon University, the Chilean Participation group, the French Participation group, Harvard-Smithsonian Center for Astrophysics, Instituto de Astrofísica de Canarias, The Johns Hopkins University, Kavli Institute for the Physics and Mathematics of the Universe (IPMU) / University of Tokyo, the Korean Participation group, Lawrence Berkeley National Laboratory, Leibniz Institut für Astrophysik Potsdam (AIP), Max-Planck-Institut für Astronomie (MPIA Heidelberg), Max-Planck-Institut für Astrophysik (MPA Garching), Max-Planck-Institut für Extraterrestrische Physik (MPE), National Astronomical Observatories of China, New Mexico State University, New York University, University of Notre Dame, Observatário Nacional / MCTI, The Ohio State University, Pennsylvania State University, Shanghai Astronomical Observatory, United Kingdom Participation group, Universidad Nacional Autónoma de México, University of Arizona, University of Colorado Boulder, University of Oxford, University of Portsmouth, University of Utah, University of Virginia, University of Washington, University of Wisconsin, Vanderbilt University, and Yale University.

Dedication

I dedicate this work to my wife and kids, who supported me with it for a significant portion (and in some cases all) of their lives.

Contents

List of Figures

List of Tables

Abbreviations

ALMA	Atacama Large Millimeter Array
API	Application Programming Interface
ASKAP	Australian Square Kilomter Array Pathfinder
AVIRIS	Airborne Visible / Infrared Imaging Spectrometer
BOSS	Baryon Oscillation Spectroscopic Survey
CARTA	Cube Analysis and Rendering Tool for Astronomy
CCD	Charge-coupled device
CRS	Coordinate Reference System
DEC	Declination
ESA	European Space Agency
FITS	Flexible Image Transport System
GB	Gigabyte
GIS	Geographic Information System
HDD	Hard Disk Drive
HDF4	Hierarchical Data Format version 4
HDF5	Hierarchical Data Format version 5
HDFS	Hadoop Distributed File System
HDS	Hierarchical Data System
HEALPix	Hierarchical Equal Area isoLatitude Pixelation of a sphere
HETDEX	Hobby-Eberly Telescope Dark Energy Experiment
HiPS	Hierarchical Progressive Survey
HiSS-Cube	Hierarchical Semi-Sparse Cube framework
HPC	High Performance Computing
I/O	Input / Output
ISO	International Organization for Standardization
IVOA	International Virtual Observatory Alliance
JSON	JavaScript Object Notation
LiDAR	Laser imaging, Detection And Ranging
LSST	Large Synoptic Survey Telescope
LSST	Large Survey of Space and Time
MaNGA	Mapping Nearby Galaxies
MB	Megabyte
MIP	Multum In Parvo
ML	Machine Learning

MPI	Message Passing Interface
MPIO	Message Passing Interface - Input / Output
MRO	Murchison Radio-astronomy Observatory
MWMR	Multiple Writer Multiple Reader
NASA	National Aeronautics and Space Administration
netCDF	Network Common Data Form
PB	Petabyte
POSIX	Portable Operating System Interface
RA	Right Ascension
SDSS	Sloan Digital Sky Survey
SFR	Star Formation Rate
SKA	Square Kilometer Array
SPLAT-VO	Spectral Analysis Tool
SSD	Solid State Drive
STEMOC	Space-Time-Energy-Multi-Order-Coverage
STMOC	Space-Time-Multi-Order-Coverage
SWMR	Single Writer Multiple Reader
TB	Terabyte
TOPCAT	Tool for OPerations on Catalogues And Tables
VFD	Virtual File Drivers
VO	Virtual Observatory
VOTable	Virtual Observatory Table
WCS	World Coordinate System
ZTF	Zwicky Transient Facility

Chapter 1

Introduction

We live in the era of Big Data - regardless of the field of study the volumes of both real and simulated data are not only growing in size but also in dimensionality. Real data coming from detectors also introduce uncertainty in the measured values, often presented as a combination of random and systematic errors. Without quantifying this uncertainty, the measured values cannot be trusted. Petabyte-scale (PB-scale) databases are expected not only to be used to store the data but also to provide it in a standard format to scientists for quick analysis. This scale of data size is still expected to be stored on traditional HDDs because of the much better cost per GB ratio and this is the reason why we should store the accessed data in contiguous blocks to minimize seek times. However, minimizing the number of read operations is relevant for SSDs as well, since the sequential reads are also faster than random accesses. A contiguous block means a data storage block that can whole be read out by one sequential read operation, regardless of whether it is stored on HDD or SSD.

With the new "Transient Universe" trend coming now in astronomy with the petascale surveys, such as Legacy Survey of Space and Time (LSST) [44] of Vera C. Rubin Observatory, or Zwicky Transient Facility (ZTF) [5] of Palomar Observatory, there comes a new level of specialization, where the dedicated survey telescope produces shortly after the exposure a high rate of transient events, each relevant for a different kind of follow-up observation. This introduces a need to combine big data not only from different detectors of the same type but also from different types of detectors. For example, the type of a detector can be a CCD camera, a spectrograph, a LiDAR sensor, or a gravitational-wave detector, anything that measures data about an object or a phenomenon that influences the object's behavior. Combination of all the data available enables higher quality analysis of the observed object.

Examples of such instruments are The Rubin Observatory Large Synoptic Survey Telescope (LSST) camera producing images, see Fig. 1.1, the Hobby-Eberly Telescope Dark Energy Experiment (HETDEX) producing large amounts of spectra per each pointing, see Fig. 1.2, Gaia satellite producing even bigger amount of spectra continuously, see Fig. 1.3, or Square Kilometer Array (SKA) instrument producing hundreds of terabyte

multi-dimensional image cubes, see Fig. 1.4. The SKA survey is at the time of the writing of this article called the "ultimate big data challenge" and is expected to run operations for another 50 years. All of these instruments are cutting-edge technologies, for example, the LSST survey is expected to start operation (first light) in July 2024 and SKA in 2027, but by that time every software for their analysis needs to be already developed and tested on simulated data.

Because of the huge data volumes involved, the exploration is too time-consuming to be done interactively. This will lead to the need to augment the exploration process with machine learning algorithms. These, however, have often different requirements on data storage optimization than interactive visualization algorithms. The only possible approach for these amounts of data is to move computing close to the data [39]. It is also convenient to use containerization to provide user-controlled environments for the scientific analysis on the server [59]. While experiments are conducted in moving the data around [28], performing the computation physically close to data avoids unnecessary waste of resources.

Figure 1.1: LSST Camera - the world's largest camera with 3.2 Gigapixel images. This camera is expected to produce approximately 15 PB of data during its 10-year lifespan. With the ability to scan the whole sky in three days time, it produces a 10-year "movie of the universe", including 20 billion galaxies, 17 billion resolved stars, and 6 million orbits of solar system bodies.

There are two major goals we aim to achieve:

1. **Fast visualization** of combined multi-dimensional big data.

2. **Fast machine access** to combined multi-dimensional big data.

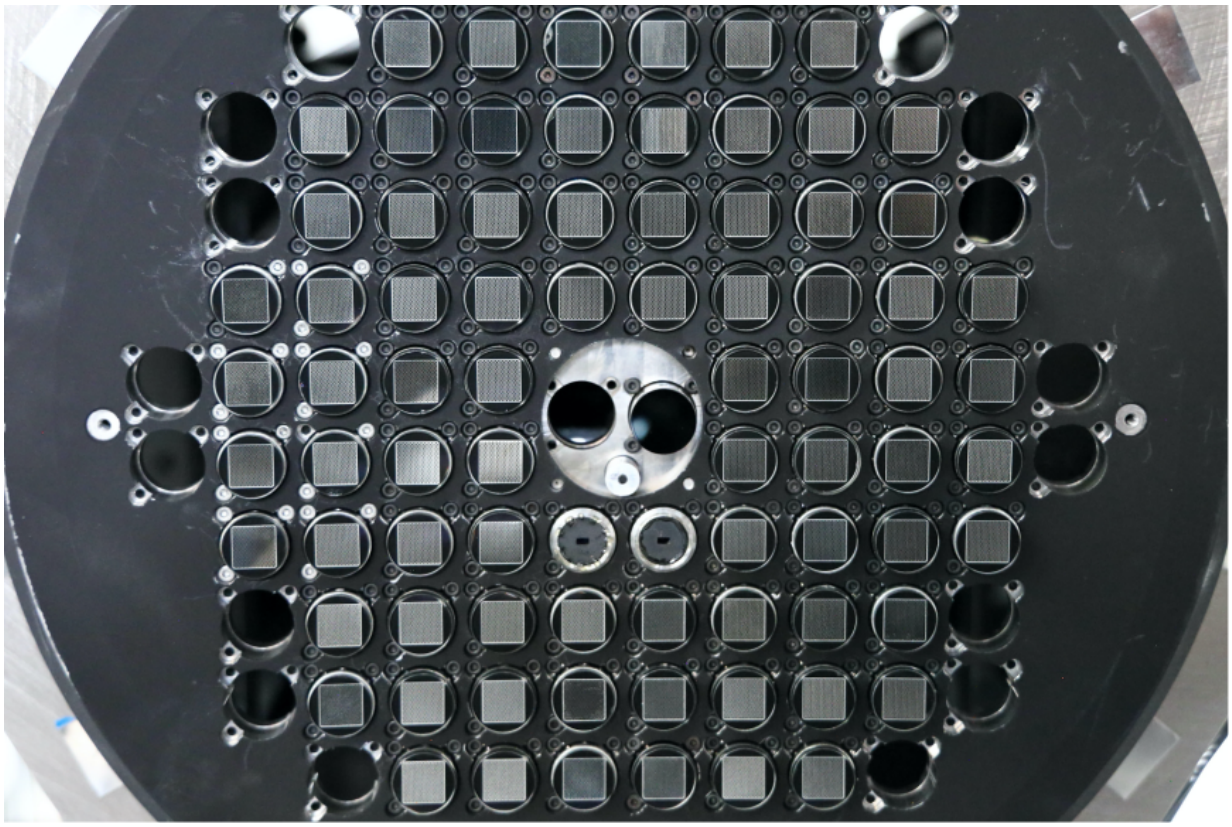

Figure 1.2: The focal plane of the Hobby-Eberly Telescope. Starlight hits an array of 78 fibre plugs (each containing 448 glass fibres) instead of a camera. These guide the light to over 150 connected spectrographs, where it is analyzed. For each pointing, the telescope can record around 32,000 spectra, simultaneously, capturing the cosmic fingerprint of the light from every object within the telescope's field of view.

Definition 1.0.1. Multi-dimensional is data that has more than two dimensions.

While we design the solution to support any kind of multi-dimensional (more specifically **dimensionally multi-modal**, see Def. 1.0.3) data, we demonstrate its capabilities within this thesis on the example of spectra and images from the Sloan Digital Sky Survey (SDSS) [6]. The dimensions of such data are spatial, spectral, and temporal, which results in a 4D data cube (spatial dimension is described by right ascension and declination) when combined. In our research of potential systems that would help us solve the goals mentioned above, we focused particularly on solutions in astronomy and Earth sciences. We added the Earth sciences because the big data are very similar in this discipline and both astronomy and Earth sciences have been producing big data for tens of years already, making it more likely to find mature solutions in these. The data is similar because observations from both are projected on a sphere on which astronomers are looking up and earth scientists down. We focused on remote sensing and climate data that have very mature and stable solutions in place. We have not reviewed the solutions for other disciplines, such as

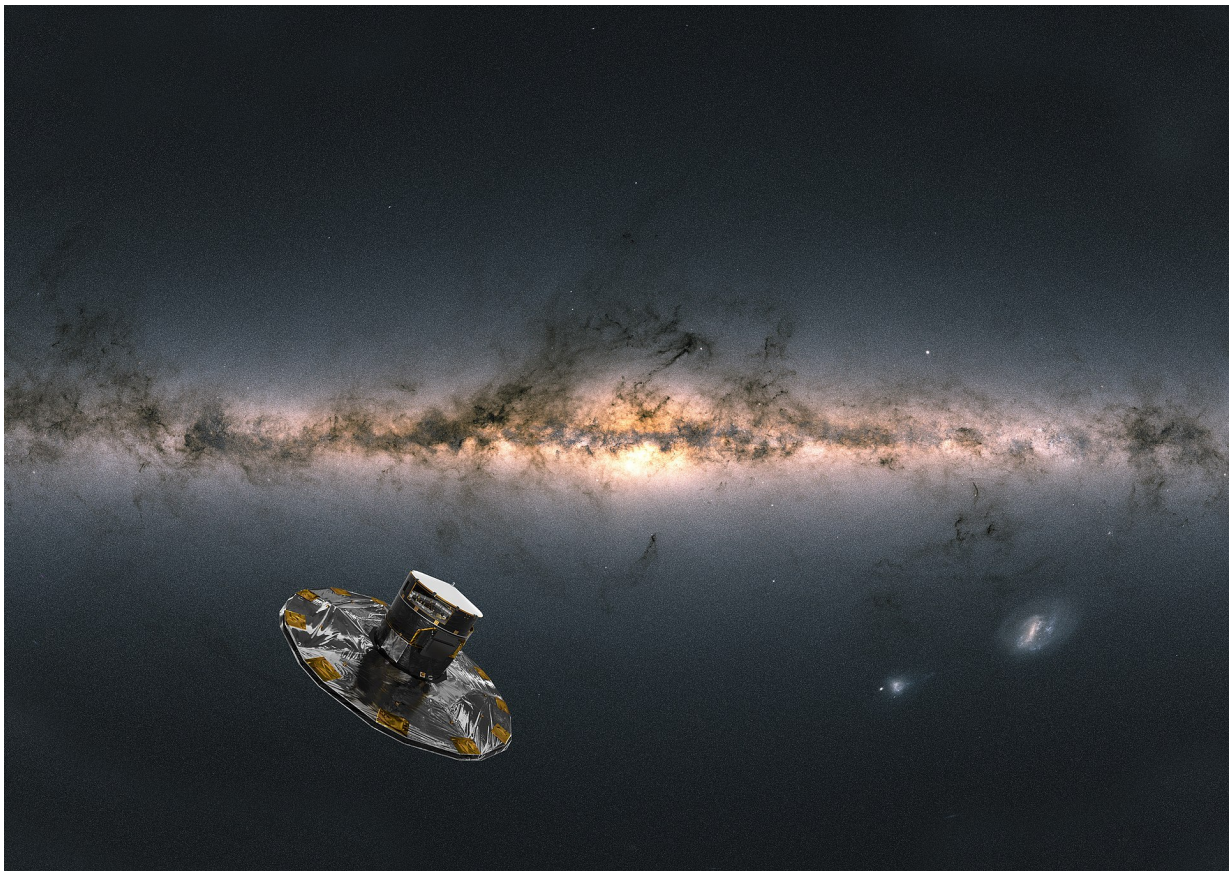

Figure 1.3: The GAIA satellite produces large spectroscopic surveys. The 3rd data release from 2022 contained spectra of 219 million light sources, making it another good candidate for combining with other imaging surveys, such as the LSST, to gain additional insights about the objects being observed.

physics, biology, or chemistry, because they have very different data, and leveraging their solutions would be very difficult, however, we design our solution so that it is extensible to these kinds of data as well.

An important feature of the visualization, which we need to implement, is an old and well-established MIP map[1] technique [69]. This technique can be used not only for images but also for any high-resolution data. A MIP map is a precomputed lower resolution of an image or of a dataset of higher dimensionality, for example, a spectral data cube. The MIP stands for "multum in parvo", meaning "many things in small place". The Hierarchical Progressive Survey (HiPS) [16] is a modern method to implement MIP maps for image visualization purposes in astronomy. However, it does support the uncertainty of measured values and its propagation to lower resolutions. It also does not support the combination of desired dimensions (spatial, spectral, and time). The filesystem-based structure of the HiPS is also focused rather on visualization than providing a database able to query and

[1]The term MIP map comes from early 3D computer graphics.

Figure 1.4: Australian Square Kilometer Array Pathfinder (ASKAP) antennas at the Murchison Radio-astronomy Observatory (MRO) in Western Australia. Once fully operational, two largest radio telescopes in the world, located in Australia and South Africa, will start to produce data in 2027. The SKA will archive 300 PB of data per year. Professor Peter Quinn of the Centre for Radio Astronomy Research (ICRAR) stated that, "This telescope will generate the same amount of data in a day as the entire planet does in a year. We estimate that there will be more data flowing inside the telescope network than the entire internet in 2020."

combine the data.

There are many well-proven specialized solutions for big data visualization, querying, and analysis in astronomy (SPLAT-VO [53], TOPCAT [60], Aladin [7], ESA sky [40]) or in Earth sciences remote sensing (ERDAS Imaging software[2], ENVI[3], ArcGIS [36]). Most of these tools in astronomy are based on Virtual Observatory principles[4]. These tools are used to analyze one type of data produced primarily by one type of detector or to analyze fused data on the same coordinate grid, such as Coordinate Reference Standard [23] for remote sensing data constructed by data fusion [49]. To the best of our knowledge, there

[2]https://gisgeography.com/erdas-imagine/

[3]https://www.l3harrisgeospatial.com/Software-Technology/ENVI

[4]https://www.ivoa.net/

is currently no efficient software for combining different kinds of multi-dimensional multi-modal big data with incompatible dimensions or sparsity.

From a computer science point of view, photometric surveys in astronomy are multi-dimensional data. Images are captured at different times and in different colors (spectral dimension). Spectroscopic surveys provide also multi-dimensional data because spectra (which are one-dimensional arrays) are measured at different times and have different spatial coordinates in the sky. More complex examples of similar multi-dimensional data are spectral data cubes [50] that have the same dimensionality (spatial, spectral, and temporal) but are less sparse than individual spectra. Another example similar to these astronomical spectral data cubes are multi-spectral or hyper-spectral images [52] produced by remote sensing detectors.

Multi-modality can either mean incompatibility of the physical quantities measured within the same multi-dimensional space (in LiDAR data the value of a pixel is the distance in meters, whereas, in images, the value of a pixel is light intensity) or incompatibility of dimensions themselves (spectra have light intensities measured for thousands of wavelengths but images are captured in five wavelength filters).

Definition 1.0.2. Data multi-modality is incompatibility of the physical quantities measured within the same multi-dimensional space.

Definition 1.0.3. Dimensional multi-modality is incompatibility of the dimensions of the same measured physical quantities.

Because *data multi-modality* point of view can be solved by data fusion [49], we focus on *dimensional multi-modality* in this thesis. We demonstrate the solution on data produced by photometry cameras and spectrographs, which both produce 4D data from a computer science perspective (spectral, temporal, and 2D spatial dimensions). Some astronomical surveys can also produce polarization as the 5th dimension. The interpretations of these dimensions are not mutually compatible. An image from a photometry camera is typically captured in several colors (using wavelength filters), whereas the spectrum from a spectrograph has thousands of spectral coordinate values corresponding to individual wavelengths. A camera collects photons from a 2D area in the sky into a 2D array of pixels in several filters; therefore, it focuses on the spatial dimensions, whereas a spectrograph aggregates all photons from a small 2D area in the sky (typically represented by the projection of the spectrograph slit or fiber size) into a single point, focusing on the spectral dimension.

Definition 1.0.4. Combination of dimensionally multi-modal data brings together data belonging to the same observed/measured object to enable analysis of all available data about that object at once.

For example, an **object** can be an astronomical object observed in the sky, such as a star, a galaxy, or a quasar.

Definition 1.0.5. Two different observations are **relevant** to each other if they capture the same object.

Combination of spectra and images means that we can analyze all measured spectra and images of one object together.

Data combination and data fusion from multiple instruments is an important problem, and some of the scientific instruments are specifically designed to produce already fused data. In remote sensing, these are multi-spectral and hyper-spectral images produced by imaging spectrometers, such as AVIRIS [63]. In astronomy, these are spectral cubes produced by integral field spectrographs, such as MaNGA [73], or data cubes from radio telescopes, such as ALMA [71]. These instruments make it much easier to work with fused data because the data cubes produced by them share the same spatial dimensions. However, these hybrid instruments have several drawbacks. They do not solve the problem of combining data from more than two types of instruments. Furthermore, specialized spectrographs or cameras always produce more precise data (higher resolution, wider spatial coverage, higher signal-to-noise ratio).

Therefore, the second major problem we aim to solve is all-to-all combinations of dimensionally multi-modal big data from any kind of detectors that produce independent measurements. The original data from the respective detectors are usually **dense**.

Definition 1.0.6. Dense data are data that do have a measured value for each combination of coordinates of their coordinate system.

Measuring dense data is a common characteristic of modern detectors, which collect data from a distance. Usually, the result of such a measurement is not a single value, but a set of values. The coordinate system of a detector has typically dimension sizes equal its resolution. The data volume per measurement using specialized detectors varies greatly, ranging, for example, from 6 GB images that need 15-second exposure in the Large Synoptic Survey Telescope (LSST) camera [44] to hundreds-of-KB spectra that need 45-minute exposure in the Baryon Oscillation Spectroscopic Survey (BOSS) spectrograph [54]. This means that the amount of observed data and its rate of acquisition varies greatly; therefore, the combined data will be also irregular in data volume.

Dense data measured by a single detector are usually stored as a dense dataset in a file system. However, if these data are projected into their original multi-dimensional coordinate system (spatial, spectral, and temporal), the data become **sparse**.

Definition 1.0.7. Sparse data are data that do not have a measured value for most of the coordinate combinations of their coordinate system.

This is the case even for detectors that produce fused data. For example, if hyperspectral images that have a measurement for each wavelength and each pixel, are projected on a map of the Earth, they will not cover the entire Earth's surface. The number of not-covered areas depends on the survey. An astronomical or remote sensing survey is a planned set of observations covering an area in the sky or on Earth. Data within the covered areas are **dense**, as is the case for "all-sky" astronomical surveys, where the covered area is the whole sky. This means the fused data has bigger dense substructures (one for each measurement), but overall the data is still sparse.

If we combine a sparse astronomical spectroscopic survey and a dense all-sky photometric survey, the resulting coordinate space will be sparsely covered (much more sparsely than in the case of a survey producing fused data). While many objects can be measured on a single image, only a relatively small number of objects can be measured using a spectrograph. Therefore, in a physical coordinate space, the data would form a very sparse matrix with dense block substructures formed by images and spectra. There are two extreme approaches to storing such data:

1. Coordinate format: data for every pixel are stored in a table whose entry consists of a measured value and its coordinates (2D spatial, spectral, and temporal). It is also possible to construct a database index of the coordinates to enable faster queries on the data. This format is most efficient for storing sparse data without dense blocks.

2. Dense block format: data are stored as individual files with one file per measurement. Each file contains metadata in its header, which enables the computation of pixel coordinates for the data within that file. The metadata are also imported into a relational database to enable faster queries on the data. However, the database contains only paths to the data files and not actual data. The efficiency of this format grows with the size of the dense blocks within the sparse data.

Storing data in these formats has the following drawbacks:

1. Storing coordinates together with the data quickly becomes very memory-inefficient with higher dimensionality data. The coordinate table stores duplicate values. For example, all measured values in a spectrum share the same spatial coordinates.

2. The dense block format is efficient for constructing cross-matched tables but not for retrieving the combined data. We explain why this is the case below.

Definition 1.0.8. Cross-matching of spatial datasets identifies and compares objects in two or more datasets based on their positions in the same spatial coordinate system [30]. In this thesis, we also use the term **linking** as a synonym for cross-matching. The cross-matching links the data from different instruments together, it does not **combine** all the available data into one coherent dataset.

On the other hand, the result of such a cross-matching query will be a table of pairs of matching object IDs; it will not contain cross-matched data. To obtain the data from the original file for each cross-matched pair, a random read from a file or an individual web service request needs to be made; therefore, it would be very slow.

In astronomy, cross-matching is usually done to match different observations of the same object based on their coordinates in the sky. The Dense block format (2) comes often with a relational database next to it that stores metadata about the objects. This relational database then stores links to these dense blocks, such as external files on a filesystem, connecting the metadata to its data. Therefore, the cross-matching of spatial

datasets stored in Dense block format is usually very fast, because it works only with the metadata that is inherently small and very well structured.

Because of these drawbacks, the data cannot be treated as either **sparse** or **dense** because storing and processing them would be inefficient. They need to be treated as **semi-sparse**.

Definition 1.0.9. Semi-sparse data is a combination of dimensionally multi-modal data.

1.1 HiSS-Cube framework

Let us now introduce the **Hierarchical Semi-Sparse Cube framework**, shortly **HiSS-Cube**, which was designed and developed in this thesis as a solution to the problems described above.

The HiSS-Cube is designed to combine dimensionally multi-modal data from different instruments. It is designed to support both human-based interactive analysis and machine learning applications. Machine learning applications need to go through the entire data set multiple times sequentially to train their models on it. HiSS-Cube is designed as a server-side framework for combining multi-dimensional multi-modal multi-resolution data.

The term **Semi-Sparse** stands for the semi-sparse data (see Def. 1.0.9) in the Hierarchical Semi-Sparse Cube (HiSS-Cube) name of the framework.

The **Hierarchical** aspect of the HiSS-Cube is two-fold. First, it uses hierarchical indexing (specifically the `HEALPix` for the spherical tesselation of spatial coordinates [19]) to enable fast queries on the data. Second, it produces **down-sampled** resolutions (also called image down-scaling), similar to GoogleMaps when zooming into the data, to enable fast visualization and machine access to the data (the first example of down-sampling was also called MIP mapping, as mentioned earlier).

Definition 1.1.1. Down-sampled data are data that have lower resolution than their original and were produced by interpolation of the original data values.

HiSS-Cube can be used as a framework for combining any kind of dimensionally multi-modal data at multiple resolutions. Let us now explain what the data combination means on the example of spectra and images. For different types of data coming from different types of detectors, similar functionality will be needed.

To achieve the goals from page 2 for spectra and images, we need a database that links the spectra with all relevant image regions that are large enough to capture the shape of the object in the sky.

Definition 1.1.2. Image region is a small area on the image that is cut out by a spatial query. Typically it is an area that is big enough to guarantee that the whole object is captured but small enough to not contain other objects.

A database query run on this database then needs to combine the linked data together into one coherent dataset that can be analyzed. This means that HiSS-Cube needs to take

spectra with thousands of spectral coordinate values and combine them with images taken in at most five different colors. It results in a dataset that is irregularly sparse along the spectral and spatial axes.

Although we focus in this article on the combination of spectra and images, HiSS-Cube can be used for combining any kind of dimensionally multi-modal data in multiple resolutions. Since astronomy was one of the first disciplines that needed to solve problems connected with analyzing big data, there are many examples of other disciplines reusing astronomical solutions. These are for example spatial profiling of cancer tissue [58] or the AstroPath [3] (Astronomy combined with Pathology) platform which is aimed to be an atlas of cancer cells. AstroPath combines mosaics of multiplex images[5] which is a problem solved in astronomy several decades ago. However, the data volume is growing in digital pathology as well and a farm of automated microscopes can produce several PB per year, and scalability is the main concern of modern systems, similar to astronomy. In fact, these solutions are based on implementations that were originally designed for SDSS data, which is the same data we use for demonstrating the capabilities of HiSS-Cube.

There are already science platforms, such as SciServer [57], that host big data from multiple disciplines. The SciServer also uses the HDF5 format for many of these disciplines.

Regarding big data, both astronomical and remote sensing data are characterized by a rather low number of dimensions compared to biology [72] or chemistry [48]. These are typically spatial, spectral, temporal, and polarization dimensions [75]. Having four to five dimensions is not considered complex nowadays, but the astronomical imaging surveys are often of the PB-scale and cover trillions (10^{12}) [24] of individual measurements. Therefore, the challenge is to scale the data combination to petabytes (PBs).

Choosing the right database or data format as a foundation for HiSS-Cube depends very much on the nature of the data. While there are proven specialized approaches for either dense or sparse data cubes, these approaches do not apply well to multidimensional semi-sparse data usually produced by detectors [37]. Visualization requires to construct optimized storage of dimensionally multi-modal data cubes created from combined semi-sparse data from different types of detectors.

In addition to visualization, HiSS-Cube needs to support optimized data access for machine processing of the combined semi-sparse data, such as machine learning or data mining. The key requirement for such machine processing is that it typically needs to access a significant portion of the combined data, so indexing is not as important as being able to very efficiently combine pairs of relevant observations from multiple detectors. The machine access can be used for example by clustering or classification of objects based on their flux and its variance in different spatial, spectral, and time coordinates. Another more complex scientific application would be a prediction of properties of the object observed by both detectors, where both detectors provide additional information (detailed chemical composition of the observed object from spectroscopic data and shape of the object from imaging or photometric data). Such property for astronomical objects

[5] **multiplex** is a synonym for hyper-spectral images or spectral cubes in digital pathology

could be Star Formation Rate[6].

For reliable results of both visualization and machine processing, HiSS-Cube needs to store the uncertainties with the data and also to enable efficient hierarchical access to the data without the need to precompute it on the fly.

All of this translates to our ultimate goal to optimize reads needed for the semi-sparse data combination both in terms of the number of I/O operations and the amount of data accessed.

1.2 Contributions of the thesis

Let us now enlist the contributions of this thesis:

- We have developed the concept of semi-sparse data as a combination of dimensionally multi-modal data.
- We designed and developed the HiSS-Cube framework that supports the combination of semi-sparse data up to the PB-scale efficiently.
- We created a HiSS-Cube File structure based on HDF5 that scales with the number of I/O nodes of the underlying parallel file system.
- We adapted the HiSS-Cube framework to support additional preprocessing strategies implemented as Python modules. Therefore, adding support for other types of dimensionally multi-modal data from astronomy, bioinformatics, chemistry, physics, and Earth sciences, is performed by implementing another Python module within the HiSS-Cube framework.
- We tested the scalability of the HiSS-Cube framework to hundreds of TB of data from the Sloan Digital Sky Survey (SDSS) [6]. The whole SDSS survey can be processed in approximately one hour.
- We performed an extensive analysis of the performance of the HiSS-Cube framework with semi-sparse data on the `Lustre` [66] parallel file system.
- We developed a standardized metadata annotation that enables seamless interactive visualization of pairs of spectra and image regions relevant to those spectra.

1.3 Thesis structure

The thesis is organized as follows.

We describe the requirements on HiSS-Cube in Sec. 2.1. Subsequently, we point out the most difficult problems connected with these requirements in Sec. 2.2.

[6] `https://www.sciencedirect.com/topics/earth-and-planetary-sciences/star-formation-rate`

In Chap. 3 we review the latest solutions to similar problems, explain why they do not satisfy our requirements, and choose a foundation for HiSS-Cube.

Then we describe the HiSS-Cube overall architecture in Chap. 4, its sequential part in Chap. 5, and its parallel part in Chap. 6. The architecture chapters also include the design of data structures used within HiSS-Cube.

The interesting parts of the implementation are described in Chap. 7. There are sections for overall, sequential, and parallel parts of HiSS-Cube implementation, followed by Sec. 7.4 which describes how the modular architecture can be extended to other types of data than spectra and images.

The results of the visualization are depicted within Chap. 8.

The performance tests are split between sequential and parallel parts into Chapters 8 and 9, respectively. The former also includes performance tests that identified bottlenecks within the sequential version of HiSS-Cube and were used for optimizing architecture of the parallel version.

The conclusions are formulated in Chap. 10.

We also provide documentation of the demonstrated spectra and images preprocessing needed for their combination in Appendix A, and prototype of HiSS-Cube on GitHub in Appendix B.

Chapter 2

Requirements and Problem definition

This chapter describes the requirements needed to be fulfilled by HiSS-Cube in order to support the goals of visualization and machine access on top of combined dimensionally multi-modal big data. We further describe the problems connected with these requirements and their implementation and discuss how they can be solved.

2.1 Requirements

2.1.1 Functional requirements on HiSS-Cube

Functional requirements, in the order of importance.

1. Combining dimensionally multi-modal data - Data need to be combined from different types of instruments in the same units and the same scales. For example, combining all spectra with all their relevant image regions.

2. Efficient storage of semi-sparse data - For the example of spectra and images, the spatio-spectral coordinate space will be covered very sparsely by the data since non-fused data do not have a spectrum for each pixel of their images. The temporal dimension is also usually sparse. However, since the images will still represent the vast majority of the dataset volume, they cannot be stored in a sparse manner but rather as a dense dataset. This would be required for any combined data types if at least one of them is dense, regardless of whether they come from astronomical or other scientific instruments.

3. Hierarchical access - Vast amounts of data need to be explored quickly to find regions of interest without the need to read data in the original resolution. This involves exploring not only the image regions and zooming along the spatial axis, but also slicing and zooming along the other axes, such as spectral and temporal.

4. Uncertainty support - Uncertainty needs to be stored along with the data to be able to display it without the need to compute it on-the-fly. Uncertainty is usually stored

as a parameter of a probability distribution, in the simplest case of the Gaussian distribution as pairs of mean and sigma values.

2.1.2 Non-functional requirements on HiSS-Cube

Non-functional requirements, in order of importance.

1. Scalability - HiSS-Cube needs to be scalable to LSST volumes (tens of PBs) to cover the images available in the fast storage for every data release [24] plus the spectra from chosen follow-up spectroscopic observations. Scalability to SKA volumes (hundreds of PBs) will be required in the future, but such scalability is expensive and difficult to test on nowadays HPC systems; therefore, we define it as an optional requirement at this stage.

2. Flexibility - HiSS-Cube needs to be flexible for any kind of data query to support different research purposes. This means that in addition to combination of spectra and images, it needs to support also combination of any multi-dimensional data (such as image data cubes or spectral data cubes) in the same way.

3. Modularity - By using industry-standard and well-supported technologies rather than a fully tailored solution to our data, we want to make sure that HiSS-Cube is easily modifiable and further extensible to other instruments and other sciences that need to combine dimensionally multi-modal data. HiSS-Cube needs to support any dimensions or types of dimensionally multi-modal semi-sparse data.

2.2 Problems

These are the key problems that need to be solved in order to fulfill the requirements stated in the previous section, ordered by their difficulty.

2.2.1 Scalability

The HiSS-Cube needs to scale to PB data. Because the computational complexity will not be very high, the bottleneck will be the file system. This translates to the ability to scale horizontally across thousands of physical disks. This is the most difficult requirement and requires implementation on an HPC with a parallel file system.

While it is easy to parallelize the processing of dense data, it is hard to parallelize a combination of dimensionally multi-modal data, where it is unknown in advance how much data from one instrument is relevant to the data from the other one. While some spectra might have hundreds of images that overlap them, some might have only a few or none. They also might have hundreds of images overlapping them in one color, but another color might be missing completely because the filter was never used for this region of the sky. Uniform distribution of irregular workloads is difficult in general.

To solve this problem and implement scalable parallel database construction and querying within HiSS-Cube, the following three subproblems need to be solved.

1. Fully parallel construction of the database index for spatial, spectral, and temporal dimensions.
2. Fully parallel and efficient linking of spectra to their overlapping image regions.
3. Fully parallel database queries that combine the data and efficient storage of their results in the database.

2.2.2 Combining dimensionally multi-modal data

Combining imaging and spectroscopy measurements can be visualized together, providing more insight into their relationship. An example of such a visualization is a sparse data cube illustrated in Fig. 2.1.

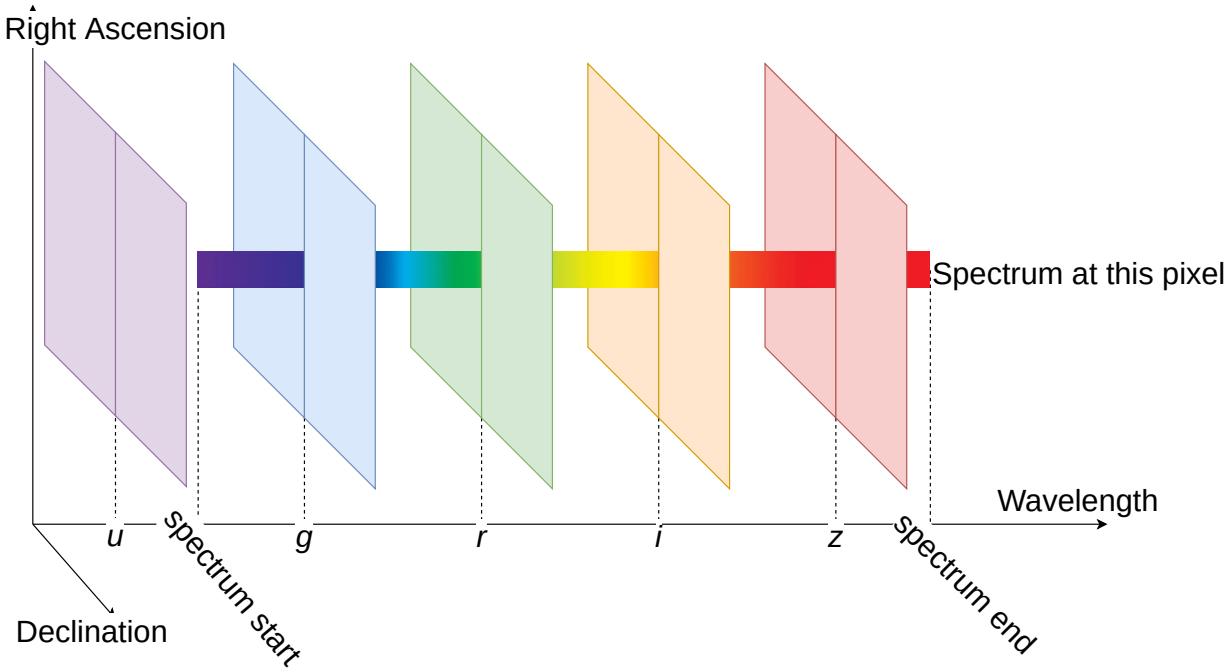

Figure 2.1: The schema of the data cube combining spectra and images. This schema represents five images and one spectrum. All of the images overlap each other spatially and are taken in five different filters.

Another problem that can also be solved by this is the ability to describe the relevance of the data from individual detectors. Using the same units for the dimensions is a necessary condition for considering the relevance of a spectrum to an image. An image further away is not relevant to a spectrum since the photons will not be gathered by the **spectrograph aperture**.

Definition 2.2.1. Spectrograph aperture is the conical area in the sky from where the spectrograph is able to collect the light, typically represented by its diameter in arc seconds.

Converting all the data to the same units and rescaling them enables also describing the relevance for the temporal and spectral axes as well. A spectrum observed a year later than the original image might not be as relevant as the one observed the same night. This relationship is illustrated in Fig. 2.2. The same goes for the wavelength relevance of an image observed in a specific filter and the spectral region on the spectrum. Also, not for SDSS data but in general, the filter curves might not have an overlap with the wavelengths observed by the spectrograph (spectra can be observed in infrared wavelengths).

While the problem of uncertainty is well addressed within its respective domains (photometry per pixel uncertainties, spectral inverse variances, etc.), when plotting such data together with these uncertainties, they will often have different units and different scales. To make a consistent data cube out of the different detectors and different detector types, the uncertainty also needs to be recalculated to common units and scales.

In our thesis, we demonstrate HiSS-Cube on producing a semi-sparse spectral data cube out of individual spectra combined with images that overlap the objects for which the spectra were measured. However, the HiSS-Cube has been designed so it is easily extensible to other data, such as Earth sciences, physics, bioinformatics, or chemistry - essentially any kind of experiments that need to combine data about one object observed by multiple detectors in a dimensionally multi-modal way.

The images and spectra are usually stored in different units and need to be recalculated. To be able to compare the values directly they need to be shifted and transformed to the same scale (photometry measured value at the given filter midpoint should be equal to the spectroscopy measured value at that wavelength). We found that the easiest way to achieve this is to use flux densities as units and apply photometry transmission curves to the covered areas of spectral wavelength. Also, the spectrograph aperture (see Def. 2.2.1) needs to be taken into account when rescaling the total flux measured by the spectrograph to the flux densities of image pixels and their "aperture" (area the pixels cover on the sky). We describe this process in more detail in Chapter 4.

2.2.3 Hierarchical Access

Propagating these uncertainties to lower-resolution MIP maps for visualization purposes is described in more detail in Section 4.1.1. The use cases for hierarchical access are described below:

- Visualization - In our opinion, the image hierarchical visualization is solved already well in astronomy. HiPS together with the Aladin tool [7] make it very easy to zoom in and out along the spatial axis. Zooming along other dimensions, such as time or wavelength, however, is not its primary use case and as such, it is not optimized for it. We did not find any other tools for hierarchical zooming along other axes than spatial.

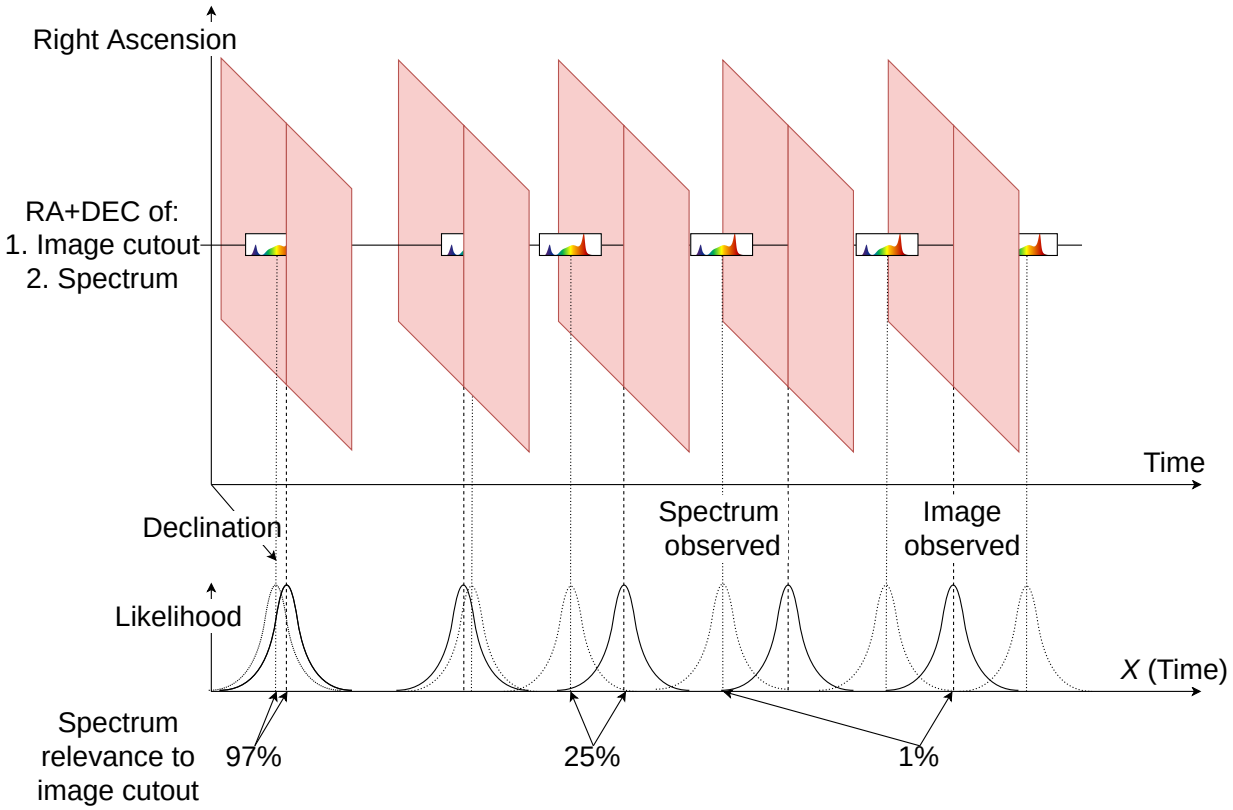

Figure 2.2: Time relevance of observations. This figure depicts a comparison of images exposed in the same filter at different time coordinates with spectra taken at different time coordinates. On the Time coordinate axis, the relevance of a spectrum can be expressed as a random variable of $X(Time)$ having a standard normal distribution. Integrating over the intersection of these two random variables will directly yield the relevance of a spectrum to an image.

- Machine learning - Machine learning algorithms are usually quite computationally intensive and retraining the model takes many hours of user's time when run on the original data resolution. What we aim for is that the retraining times are short enough so that the user can verify results almost interactively. This happens on a much lower resolution with lower reliability of the results but enables to find the potentially interesting data that can be analyzed in the original resolution afterward. Another use case can be an application of the same machine learning model, e.g., classification of galaxies, on multiple resolutions without the need to retrain the model. This could essentially classify galaxies regardless of their distance and size on the image with the same model.

To implement the hierarchical access, these subproblems need to be also solved.

- MIP map generation - There are numerous MIP mapping techniques that can be

used for the visualization [38]. Choosing the right one depends also on the type of data.

- Propagating uncertainties to lower resolutions - Once the uncertainties are calculated for the base resolution of the image or spectrum, they need to be propagated to lower resolutions properly. This is dependent on the probability distribution chosen to describe the uncertainty in the base resolution of the dataset. We chose the uncertainties to be described by single Gaussian distributions but we want the data model to be extensible to any kind of probability distributions or their mixtures.

2.2.4 Uncertainty support

There exist many definitions of uncertainty; therefore, we have decided to use the following one. We also try to clarify other related terms.

Definition 2.2.2. Measurement errors is a difference between the measured value and the true value of a measurement.

Since the true values are not usually known, much more useful information is the **uncertainty**

Definition 2.2.3. Uncertainty is the range of possible values within which the true value of measurement lies.

Although the uncertainty can be described in multiple ways, we have chosen the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean value and σ^2 is the variance. The reason to use this distribution is that it is a good approximation of Poisson distribution that characterizes both spectroscopy and photometry measurements. In some cases of high energy data, this approximation cannot be applied and the uncertainty needs to be stored as Poisson distribution; therefore, HiSS-Cube needs to support different distributions and their parameters as well.

Definition 2.2.4. Sigma (σ) is the scale for the Gaussian distribution, which equals the standard deviation in Gaussian distribution. [1]

In astronomy, or detector measurements in general, the terms **random error** and **systematic error** are often used. In terms of statistics, the random errors are similar metrics to the **precision**, and the systematic errors are similar to the **accuracy**. For the visualization of precision and accuracy, see Fig. 2.3.

Definition 2.2.5. Accuracy is the closeness of a measurement to the true value.

[1] In this thesis it is used as a synonym to the **statistical error** described in the Frame data model in the SDSS vocabulary `https://data.sdss.org/datamodel/files/BOSS_PHOTOOBJ/frames/RERUN/RUN/CAMCOL/frame.html`

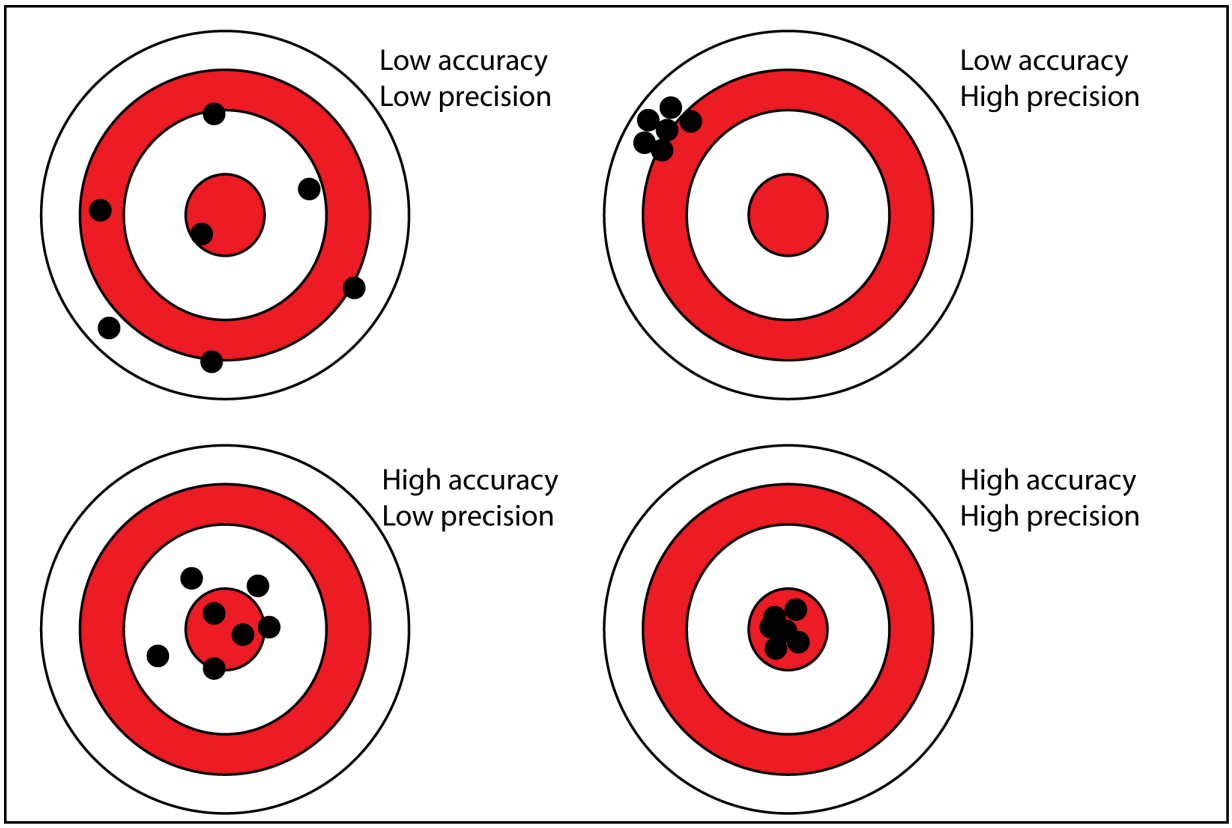

Figure 2.3: Accuracy and precision in measurements. The precision increases from left to right and the accuracy from top to bottom.

Definition 2.2.6. Precision is the degree to which repeated measurements under unchanged conditions show the same results.

To describe these together, compare them to the definition of **trueness**.

Definition 2.2.7. Trueness is the closeness of agreement between the average value obtained from a large series of test results and an accepted reference value.

When questioning whether data are true, we gauge them by this **trueness**. In HiSS-Cube, we can focus only on describing the random errors (precision) because the systematic errors (accuracy) are already the main concern of data reduction pipelines and therefore mitigated significantly.

Definition 2.2.8. Data reduction pipeline is the process that creates science-ready data from the raw observed data.

The random errors are usually much harder to get rid of but at the same time, it is much easier to describe them by a probabilistic distribution. In the example of spectra and images, we chose the Gaussian distribution.

Rapid progress in the research of Bayesian deep learning, which enhances deep learning with uncertainties on the output [17], is of the highest importance to astronomers. Papers [56] and [33] showed that meaningful uncertainties improve the usability of machine learning methods in astronomy. However, the mentioned papers take into account only uncertainties in model parameters, not the uncertainties in measurements (the ones that we define in this thesis) which could further improve the quality of the model results.

Computing uncertainty for each measurement is a process based on the physical parameters of detectors and on external observational conditions. Data providers usually do not publish the uncertainty for each measurement explicitly. They rather document the reduction pipeline to compute the measurement uncertainty from the observational metadata. However, this approach does not allow to visualize the uncertainty in data right away when exploring the data. For example, in radio astronomy imaging the uncertainty needs to be provided together with the data, since producing it is a very complicated process [27].

The problem of uncertainty needs to be solved for both visualization and machine learning purposes:

- Visualization - The problem of uncertainty visualization in astronomical data has been addressed for some time already [34], but only a few astronomical datasets provide the uncertainties together with the data. They often need to be computed as a part of the preprocessing. While this can be done and it certainly saves storage space, it makes it much harder for scientists to explore the data interactively from the *trueness* point of view.

- Machine learning - Some machine learning algorithms produce uncertainty as a part of their output. For example, an algorithm for astronomical object classification could provide also confidence intervals for each object it classifies with a given score. However, this information can be further enhanced if it is already known that the data is noisy on the input. For example, the algorithm knows that it cannot provide a confident result if the image of the galaxy it is classifying was taken on a bad night with high noise. Having the data uncertainty available in multiple resolutions at the same time can also be used by a machine learning algorithm. This has been proved within other disciplines, such as digital pathology [25].

The uncertainty for each measurement is seldomly published as a part of the data since it is considered redundant. However, the process of calculating the uncertainties can be time-consuming and therefore unacceptable for algorithms that need to go through the big dataset multiple times (for big data, the dataset does not fit into memory). The process for uncertainty extraction can also be different for each data supplier. It can be trivial when the data is assumed uncorrelated, however, it can become very complex if we take correlations into account to produce more precise uncertainty estimations [18].

The uncertainty also needs to be taken into account when aggregated or integrated data are produced, as shown in Fig. 2.4. The data integration along the spatial axes of `RA` and `DEC` happens when the lower resolution MIP map image is produced, but data can also be integrated along the `Time` axis. In that case, again uncertainties can be considered

and images captured at times with higher quality (better seeing, atmospheric conditions) will be accented while higher uncertainty values will be neglected.

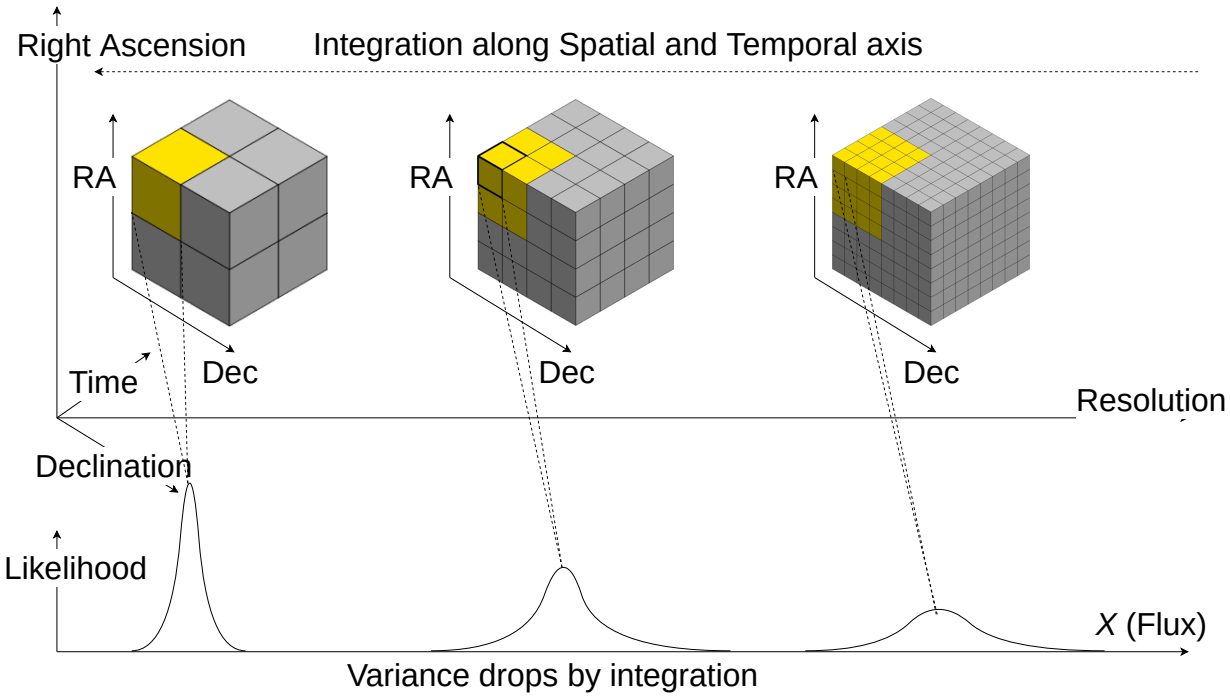

Figure 2.4: MIP maps of a 3D cube with Right Ascension, Declination, and Time axes. The original resolution images on the right have 8x8 pixels, observed at 8 different times within regular intervals. When moving left we integrate the measurements along both spatial and temporal axes, i.e., 1 pixel will be constructed from each 2x2x2 pixel region, which reduces the data volume dramatically and also increases the probability that the lower resolution value is a correct one since we put more weight on pixels with smaller measurement error.

Chapter 3

State of the art

In this chapter, we discuss the existing solutions to similar problems. We review these from the perspective of our requirements defined in Sec. 2.1.

3.1 Rasdaman array database

Rasdaman (Raster Data Manager) is an array database system [4] that provides flexible, fast, scalable geo services for multi-dimensional spatio-temporal sensor, image, simulation, and statistics data. It uses the patented datacube federation mechanism to scale to tens of PBs of Earth sciences data. It is a mature system available since 1992. It focuses on dense arrays which both the images and spectra are, at least as dense substructures within overall sparse data.

A big advantage is that Rasdaman supports many modes for tiling, including irregular tiling, which can define regions of interest. These enable optimization of I/O operations needed to access the interesting regions on disk, such as image regions relevant for spectra that overlap them. It is also essential for astronomy applications in general since even the homogeneous surveys do not have rectangular coverage and have more dense and more sparse areas.

3.1.1 Rasdaman drawbacks

We tried to implement HiSS-Cube in the database `Rasdaman`. Utilizing Rasdaman for astronomical data (or any kind of different than Earth science data) would require to overcome additional problems caused by its specialization. The differences and how they could be overcome are listed below:

1. **Density and data shape** - The Earth sciences community (or at least the implementation of their databases) heavily focuses on dense data coming from data fusion. In fact, the usual practice is to use materialized coordinate space as another dimension of the data to make the storage more generic and be able to store the data

together with their coordinates in dense file system serialization. Another reason is that the Earth sciences datasets cannot always be mapped to an orthogonal grid or its spherical projection since they use many different coordinate systems.7.4

This could be overcome by aggregating the data into one dense dataset and mapping it to a spatial or other coordinate system using an application layer.

2. **Metadata description** - The geographic Coordinate Reference System[1] (CRS) used in Earth sciences differs from the Flexible Image Transport System (FITS) [67] World Coordinate System (WCS) [20] [9] [21] [8] used in astronomy. CRS uses many reference systems. This is due to the tectonic plate shifts. For example, a reference system that is fixed to the North American plate is not valid for European coordinates, since these tectonic plates move apart by cca 2.5cm/year. At the date of writing of this thesis, this translates to almost 18000 coordinate systems maintained and provided by standardized *EPSG Geodetic Parameter Registry*[2].

   However, it would be possible to reuse an Earth sciences database by defining the CRS for an astronomy dataset where right ascension and declination reference points would be defined similarly to longitude and latitude.

3. **Format support** - The Earth sciences community focuses on using raster data but has also more general formats in wide adoption, namely the netCDF which is based on HDF4 and is HDF5-compatible. There exist converters between FITS format[3] and netCDF developed by NASA CDF group[4]. However, this conversion is not sufficient for us. It is because in an ideal case, we would like to have one HDF5 file representing the whole database containing the data from a large number of FITS files. Most of the currently existing HDF5-based data models preserve bidirectional FITS-to-HDF5 compatibility. This constraints the ability to leverage HDF5 libraries to their full extent and would not provide the performance we seek.

For our use case, the disadvantage was the Rasdaman's specialization for the Earth sciences data and a lot of work would be needed to fill it with the astronomical spectra and images, including modification of its source code. This would be the case for other types of data as well, which would impact the flexibility and modularity of the solution. Rasdaman also did not offer enough modularity to define custom indexing mechanisms that are needed to efficiently combine spectra and images.

It did not perform well with respect to the following requirements, ordered by the implementation effort needed to overcome them: *Combining dimensionally multi-modal data, Efficient storage of semi-sparse data, Flexibility, Modularity.*

[1] `https://www.iso.org/standard/74039.html`
[2] `https://epsg.org/home.html`
[3] `https://fits.gsfc.nasa.gov/fits_standard.html`
[4] `https://cdf.gsfc.nasa.gov/html/dttools.html`

3.2 TileDB array database

TileDB is a powerful engine for storing and accessing dense and sparse multi-dimensional arrays. TileDB can be used to store data in a variety of application domains, such as genomics, geospatial, finance, and more. The power of TileDB stems from the fact that any data can be modeled efficiently as either a dense or a sparse multi-dimensional array. By storing the data and metadata in TileDB arrays, all the data storage can be abstracted and any data science tool can be used to access the data.

In [46] TileDB was compared to HDF5 and it has shown good performance in writing the data randomly distributed into different chunks. A chunk is a storage unit that is guaranteed to be written to disk as a contiguous block, enabling sequential read. This would help in the initial build of the database. However, we focus on read performance primarily and TileDB has design trade-offs that make write performance better at the cost of read performance. Also, we would need to modify its source code to use it for HiSS-Cube implementation. Moreover, it lacks the ability to link together the spectra and images in a semi-sparse dataset and it also lacks the modularity to add this functionality easily.

TileDB did not perform well with respect to the following requirements, ordered by the implementation effort needed to overcome them: *Combining dimensionally multi-modal data, Flexibility, Modularity.*

3.3 SciDB + MonetDB array databases

Astronomy needed databases to analyze big data for quite some time and has pioneered array databases such as SciDB [14] or MonetDB [35]. There are also efforts to build common querying languages for the array data, such as SciQL [74].

SciDB is a column-oriented database management system designed for multi-dimensional scientific, geospatial, financial, and industrial data management and analytics.

MonetDB is an open-source column-oriented relational database management system. It is designed to provide high performance on complex queries against large databases, such as combining tables with hundreds of columns and millions of rows.

SciDB and MonetDB have both shown their feasibility for astronomical data, but both lack the support for semi-sparse data by default since their primary focus is not data combination. APIs of both systems do not allow extension of their functionality to support semi-sparse data efficiently. MonetDB, on the other hand, could be a potential alternative to the MySQL solution for LSST in both speed and scalability [51]. However, we would still need to build additional functionality to be able to link spectra and images efficiently. The user-defined functions provide an extension point for the querying language for both databases, but that is not enough, since we also need to control the data storage to make efficient links between the spectra and images.

These databases did not perform well with respect to the following requirements, ordered by the implementation effort needed to overcome them: *Flexibility, Modularity, Efficient storage of semi-sparse data, Combining dimensionally multi-modal data.*

3.4 Relational databases

Using relational databases as the backend storage for array management, such as Vertica database [32], RAM database [62] for dense arrays, and SRAM database [13] for sparse arrays, relies on an extra index and stores only non-null values in the sparse coordinate space. Often the index is materialized as an extra column of the relational database table and then the query is built on this column. However, even if the tables are clustered (sorted) on disk based on this indexing column, the performance on dense arrays is much slower in a relational database compared to specialized array storage. Furthermore, combination of the multi-modal data is not efficient within these relational databases and the data would need to be stored outside of the database. This would result in additional application logic needed to control the storage on the file system and would severely limit the modularity and flexibility of the overall solution.

Another approach used in the Terabase Search Engine [70] loaded the data into the relational database tables. However, it is efficient for querying small parts of the data only, not for all-to-all data combination.

Relational databases did not perform well with respect to the following requirements, ordered by the implementation effort needed to overcome them: *Scalability, Combining dimensionally multi-modal data, Efficient storage of semi-sparse data, Modularity, Flexibility.*

3.5 Hadoop HDFS

We reviewed the `Hadoop HDFS`, namely the Parquet data format[5] that provides good results for columnar data and is scalable [64].

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. Therefore, compared to other distributed file systems it is heavily focused on fault tolerance. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data.

The Apache Parquet is an open-source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.

However, the column-oriented data formats provide the best results with tables having many columns rather than big multi-dimensional arrays having few columns. Also, similarly to SciDB or MonetDB, using columnar storage for the data would require storing the coordinates also inside the HDFS as part of the data, which would double the size of the stored data. Such an approach is very cost-inefficient and not acceptable for PB-scale data. Moreover, implementing uncertainty as another dimension of the data would be rather complicated and inefficient.

[5] `https://parquet.apache.org/docs/`

Hadoop HDFS data formats did not perform well with respect to the following requirements, ordered by the implementation effort needed to overcome them: *Efficient storage of semi-sparse data, Combining dimensionally multi-modal data, Uncertainty.*

3.6 HDF5

The HDF5 (**Hierarchical Data Format version 5**), is an open-source data format and a high-performance software library for handling this data format. It is also sometimes called *I/O middleware.*

The HDF5 file can be a single `.h5` file that is a portable self-explanatory container, i.e., in addition to data, it contains all the metadata. Inside the file, it uses HDF5 groups which are equivalent to folders in a file system, HDF5 datasets which are equivalent to files, and HDF5 attributes that can be attached as metadata to both groups and datasets. The datasets are internally implemented as multi-dimensional arrays, which is very important for our multi-dimensional data. The maximum number of dimensions in a dataset is 32, which easily satisfies our requirements to store uncertainty along with data and to combine spectra and images (4D data). It is also flexible to add more dimensions if needed. The maximum theoretical size of an HDF5 dataset is 16 exabytes and there is no limit on file sizes. This means that HDF5 is theoretically capable of scaling to both LSST and SKA data volumes. The HDF5 also enables hierarchical structures by using groups.

HDF5 satisfies natively the following requirements: *Scalability, Uncertainty, Modularity, Flexibility* while the requirements *Combining dimensionally multi-modal data, Efficient storage of semi-sparse data* would need to be implemented. However, due to the flexibility and modularity of HDF5, this can be done, as discussed further below.

Since it supported many of the requirements, we expand on HDF5 advantages and disadvantages compared to the reviewed databases. The disadvantages are not on the list of our requirements, but we outline them anyway to show the trade-offs the HDF5 takes compared to the databases.

We also describe the advantages compared to the `Parquet` data format in a special subsection.

3.6.1 Advantages of HDF5 compared to databases

- Modularity - Comparing HDF5 to databases is not always possible directly, as it is a middleware, and as such it provides lower-level control than a database would. It offers both the flexibility and modularity to optimize by building HiSS-Cube on top of existing APIs, rather than having to extend the functionality of a database by modifying its source code. While building the HiSS-Cube functionality will require writing code anyway, in the case of HDF5 it would be built as an application layer on top of the middleware, rather than modifying the application layer of an existing database, which could cause conflicts with future updates. The HDF5 became recently a very popular middleware and many mature libraries are built on top of it,

such as `PyTables`[6] and `h5py`[7]. The `h5py` provides low-level API to use the C objects and procedures of the underlying library as well. The focus on Python interfaces is aligned very well with our requirement to bring the scientific analysis close to the data by using Jupyter Notebooks to be able to work interactively with the database on the server without transferring data to the client.

- Scalabilility - Paper [22] shows that HDF5 in a combination with the `Lustre` file system can perform very well on HPC systems. With some optimizations, it runs close to the performance limits of the underlying file system. Moreover, it maintains its performance while scaling to up to 40960 cores [22]. That said, HiSS-Cube still needs to solve the mapping of semi-sparse data to the HDF5 so that the scalability will work for such data as well. This should be solvable because in our case, the computational complexity of data combination depends on the overlapping of the data, rather than on their total volume. In the area of sequential read/writes, which are essential for the machine access goal from Page 2, every database system we reviewed is just getting close or on a similar level of performance of HDF5. Since both spectra and images are indeed dense arrays, this dense specialization of HDF5 is aligned with our use case and we identified this level of performance as a condition that removed SciDB and MonetDB from our shortlist. A detailed comparison between HDF5 and SciDB is shown in paper [46].

3.6.2 Disadvantages of HDF5 compared to databases

- Random/indexed access - A relational database enables random access and while the array databases provide lower performance in this regard, HDF5 is completely specialized for reading the data in contiguous blocks and it is efficient for reading and writing vast amounts of data sequentially. Although our primary use case is to do exactly that, the semi-sparse data needs to be transformed into this contiguous storage in the first place and that is why some kind of indexing is required. The indexing is not supported by HDF5 natively and we need to implement it because it is needed for both efficient visualization and linking of spectra and images.

- Appending/Modifying data - The HDF5 is bad in handling random writes and even worse when these updates are changing the file structure (such as creating/removing groups or datasets), as these *collective* operations need to be synchronized among all processes. However, this is not a big issue for HiSS-Cube, since the data needs to be preprocessed once per data release and then it is read many times without any updates needed. We focus on read rather than write performance.

- Query language - As the HDF5 is rather an I/O middleware than a full database, it misses the query language interface, which is usually supported by underlying indexes

[6] `https://www.pytables.org/`
[7] `https://docs.h5py.org/en/stable/`

to be able to query data efficiently. This is not important for machine learning algorithms that usually read the data sequentially but we will need to implement a substitute for visualization and interactive analysis that usually access only small parts of the data.

- Authentication - Databases usually provide some form of authentication for assuring data privacy and access control. HDF5 does not have such mechanisms by design since it is not a database. There is a possibility to encrypt the datasets or their chunks by applying encryption filters and this could be used as a basis for authentication and access control. However, this is not our interest.

3.6.3 Advantages of HDF5 compared to Parquet

- Efficient storage of semi-sparse data - The Hadoop data formats used within HDFS and HDF5 data format have many similarities. The key difference is that HDF5 is optimized for storing multi-dimensional arrays while the Parquet format is optimized for columnar storage of 2D tabular data. Although multi-dimensional data storage can be implemented in Hadoop HDFS data formats, such as Parquet, Avro, or sequence files, this inefficiency of multi-dimensional data storage puts Hadoop HDFS behind HDF5, even though it fits better to the implementation of our requirements compared to databases.

In the end, we chose HDF5 because it fulfills all our functional and non-functional requirements natively, except the *Combining dimensionally multi-modal data, Efficient storage of semi-sparse data*, which we can implement rather easily compared to other reviewed databases thanks to the flexibility and modularity of HDF5.

3.6.4 Existing HDF5 implementations

HDF5 has been used for astronomical data in porting FITS to HDF5 [47] or porting HDS to HDF5 [26]. The latest similar application is an alternative storage scheme for SKA data [12]. The purposes of these papers are slightly different though, as they primarily focus on comparisons to FITS-based storage performance or on converting a different data format like HDS to HDF5 in order to keep backward compatibility with the existing clients (such as visualization tools). We do not constrain ourselves by these conditions since there are no mature clients for analyzing dimensionally multi-modal data anyway; therefore, we can fully utilize the strengths of HDF5. Preservation of all the metadata from FITS in HDF5 is not an issue, in fact, HDF5 is more flexible because the HDF5 attributes can be attached to both HDF5 groups and datasets.

Chapter 4

HiSS-Cube Architecture

This chapter describes our design of the HiSS-Cube architecture satisfying the requirements in Section 2.1. This chapter focuses on the generic part of the architecture that is not specific to sequential or parallel implementation. These are further described in chapters 5 and 6.

The architecture is depicted in Fig. 4.1. The key components of the architecture are:

1. *Database construction* - The process that combines spectra and images and stores them in the database.
2. *Database queries & Storing query results* - The database queries run on the HiSS-Cube File and their results are stored back into the HiSS-Cube File.
3. *HiSS-Cube File* - Database of combined semi-sparse data in an HDF5 file.
4. *Semi-Sparse cube database interface* - Interface for visualization of semi-sparse data from the HiSS-Cube File
5. *Dense cube stored query interface* - Interface for machine access to the combined big data within the HiSS-Cube File

These components are further described in the subsections of this chapter. All these components are generic for any kind of combined semi-sparse scientific data (astronomy, Earth science, bioinformatics, etc.) but we describe their particular implementation details of the semi-sparse data combination for SDSS spectra and images to show the complex processing that needs to happen in these stages. HiSS-Cube is designed in such a way that all of the SDSS-specific steps can be replaced by Python modules for different types of data in a plug-and-play fashion.

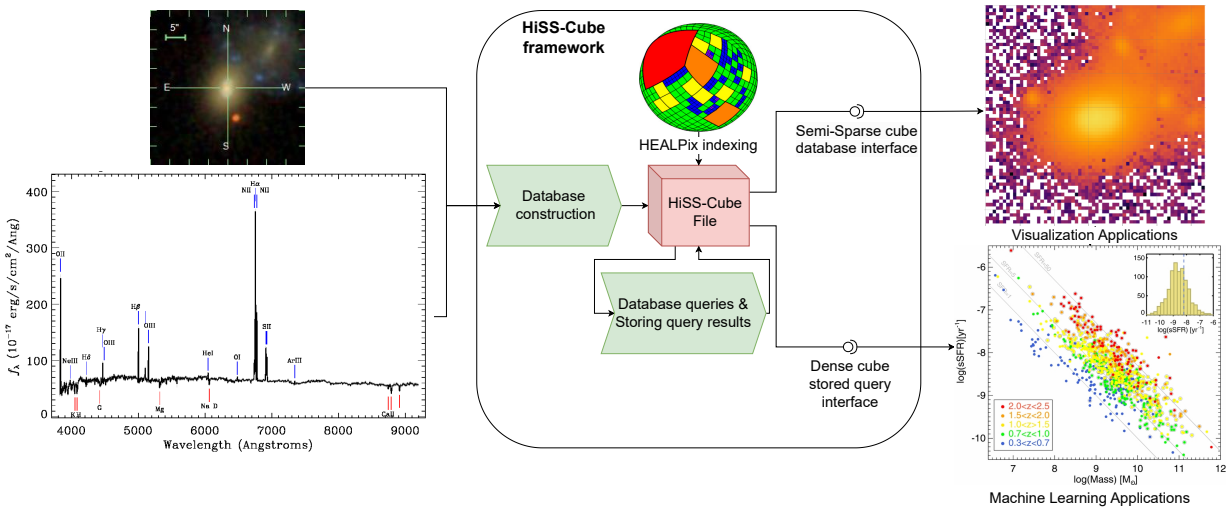

Figure 4.1: Architecture of the new version of HiSS-Cube. On the left-hand side, images and spectra in FITS files are imported into the HiSS-Cube File. The *Semi-Sparse cube database interface* can then be used for visualization of their combination in the `H5Web`[1] tool and the *Dense cube stored query interface* for machine access in machine learning applications. We use the `HEALPix` framework for spatial indexing of the data.

4.1 Database Construction

In this section, we describe the individual preprocessing steps of the database construction whose task is to enable: extracting uncertainties, calculating lower resolution, unit conversion, and spectra normalization. All these steps are needed to enable the seamless combination of spectra and image data, i.e., the *Combining dimensionally multi-modal data* requirement.

The whole complex preprocessing is shown in Appendix A and is part of the HiSS-Cube framework documentation on GitHub[2].

Let us now describe only the key steps needed to be done in the preprocessing.

4.1.1 Uncertainty extraction

For SDSS images we use the algorithm for extracting photometry uncertainties from calibrated images described in the SDSS Frame documentation[3]. For spectra, it is much easier, since there is already inverse variance stored along with the data for all of the co-added spectra[4].

[2] `https://github.com/nadvornikjiri/HiSS-Cube/blob/master/notebooks/SDSS_cube.ipynb`

[3] `https://data.sdss.org/datamodel/files/BOSS_PHOTOOBJ/frames/RERUN/RUN/CAMCOL/frame.html`

[4] `https://data.sdss.org/datamodel/files/BOSS_SPECTRO_REDUX/RUN2D/spectra/PLATE4/spec.html`

Spectra are read from FITS files unchanged in the original resolution. For images, a more complicated procedure is needed that involves several operations on each pixel of the image, including non-trivial interpolations that extract the uncertainties based on several physical attributes of the detector and the weather conditions influencing the image quality. This also illustrates the need to have the uncertainties stored along with the data since its calculation can be costly.

For the images, the calibration needs to be reversed from the calibrated image to be able to reproduce the (nearly) original data numbers coming from the SDSS CCD detectors. We presume that the number of photo-electrons collected by the CCD is a statistical quantity with the Poisson distribution. The sources of noise that affect the measured values the most are the read-noise and the noise in the dark current. These combined are put together as *dark variance* within the SDSS vocabulary. Thus, to calculate per-pixel uncertainties, the *gain* and *dark variance* values are needed. In fact, these are nearly fixed as a function of `camcol` (camera column) and `filter` attributes that are present in the image metadata. These constants also come from the documentation of the SDSS Frame. To estimate the uncertainties in the *data numbers* coming from a detector for each pixel, we used Eq. 4.1:

$$\sigma = \sqrt{D/g + \sigma_d^2} \tag{4.1}$$

where σ denotes the *standard deviation*, D denotes the *data numbers* coming from a CCD detector, g denotes the constant *CCD gain* given by a camera `column+filter` combination, and σ_d^2 denotes the *CCD dark variance* for camera `column+filter` combination. A viable visualization of the uncertainty is shown in Fig. 4.2.

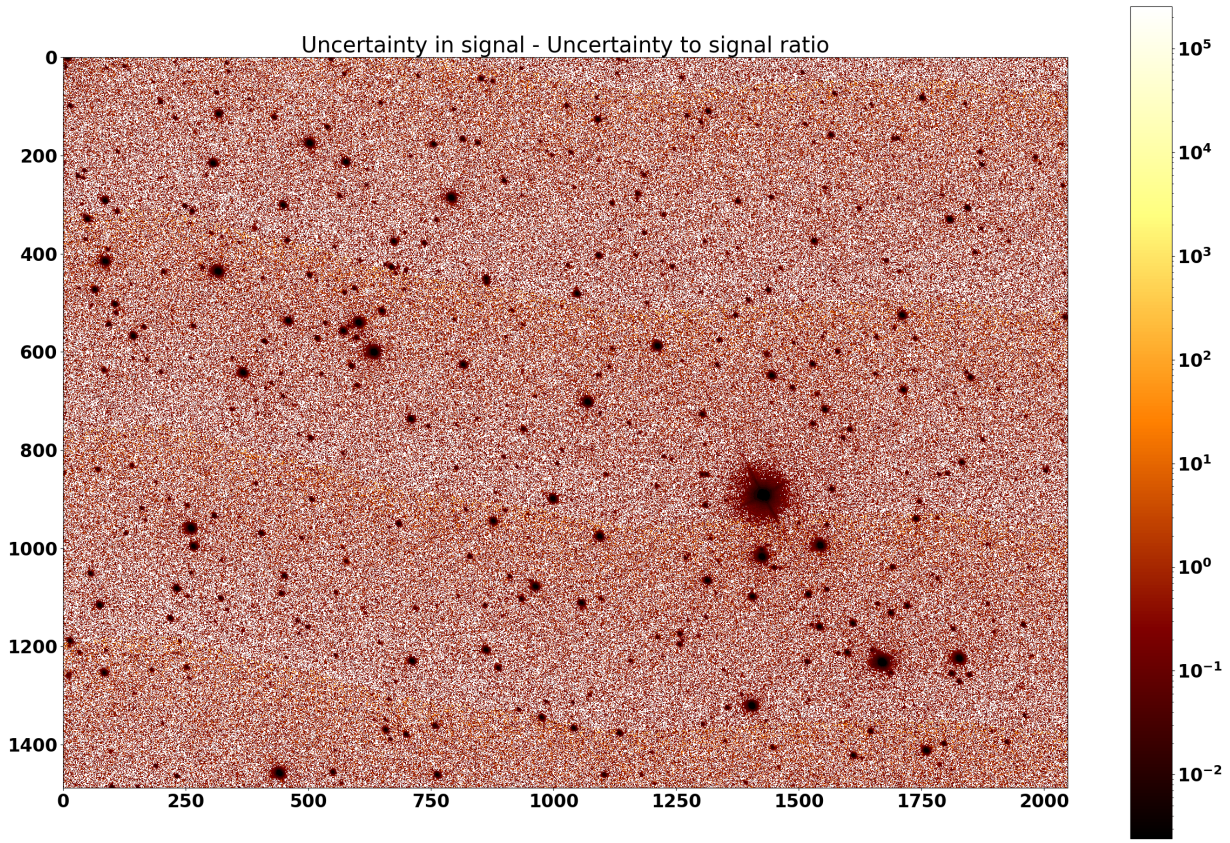

Figure 4.2: Uncertainty ratio of SDSS image. This figure depicts the uncertainty of one SDSS image as a ratio of the absolute per-pixel uncertainty to the signal strength (inverted Signal-to-noise ratio). This *uncertainty normalized by signal strength* visualizes where the measurements are less uncertain (on the bright areas, such as stars) while increasing up to 10^5 ratio in the sky background where the signal values are close to the noise values.

4.1.2 Calculating lower resolutions

For both spectra and images, we used cubic interpolation for constructing the lower-resolution pixels. For every lower resolution, the amount of pixels is halved in each dimension. Additionally, the curve of the spectrum is smoothed out since this better simulates lower-resolution spectrograph observations. This has already been proven to be a good practice when re-binning higher resolution spectra to lower resolutions for machine learning purposes [65].

Uncertainty propagation is described in detail in the ISO *Guide to the expression of uncertainty in measurement*[5]. The uncertainty u of a quantity y formed by combining measured quantities x_i defined as $y = f(x_1, x_2, .., x_N)$ is

[5] `https://www.iso.org/standard/50461.html`

$$u^2(y) = \sum_{i=1}^{N} \left(\frac{\partial f}{\partial x_i}\right)^2 u^2(x_i) + \sum_{i=1}^{N} \sum_{j \neq i=1}^{N} \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j} u(x_i, x_j) \tag{4.2}$$

where $u(x_i)$ denotes the uncertainty in x_i and $u(x_i, x_j)$ the covariance between x_i and x_j. We disregarded the correlations between measured pixel values and their uncertainties for the purpose of our thesis since the interesting features of both spectra and images are the peaks in the measured values. These can be emission and absorption lines in a spectrum or a star in a dark sky. These features are indeed not correlated, compared to the rest of the spectral continuum or the sky background. This means that the covariance between pairs of measurements is zero and Eq. 4.2 is reduced to "sum of squares" and simplifies to

$$u^2(y) = \sum_{i=1}^{N} \left(\frac{\partial f}{\partial x_i}\right)^2 u^2(x_i) \tag{4.3}$$

This final equation is used for calculating both spectrum values and image pixels for lower resolutions.

4.1.3 Converting to the same units

The SDSS spectral measurements are stored in flux densities, specifically $ergs/s/cm^2/$. We convert the images to the same units, so we can compare them directly.

We chose this flux density unit deliberately because it is very convenient for comparing multi-modal data, such as spectra and images. The flux density in common language is *energy per second per square cm per Angstrom* (1 Angstrom equals 0.1 Nanometer). Therefore, it is normalized energy in all dimensions *Temporal, Spatial, and Spectral* and allows us to compare the value measured on the spectrum directly to the value of a pixel measured on an image.

A similar approach needs to be taken in any kind of dimensional multi-modal data combination since by definition the detectors have different resolutions in the individual dimensions and these need to be normalized to make the data directly comparable.

4.1.4 Spectra normalization

To have the spectra directly comparable to images, we applied the SDSS *Filter curves*[6] to them. Applying the *Filter curve* to the spectrum simulates an approximation of the light traveling through the optical filter before it reaches the spectrograph. Since both the spectra and images are in flux densities, this needs to result in almost equal values for measured flux density on the image and at the band midpoint spectral coordinate of the spectrum. It cannot match precisely, because of random errors involved and also the fact

[6] `http://svo2.cab.inta-csic.es/svo/theory/fps/index.php?id=SLOAN/SDSS.u&&mode=browse&gname=SLOAN&gname2=SDSS#filter`

that the observed object might have changed between the image and spectrum observation times.

To make the flux density of the spectrum correct, one additional step needs to be done. Each point of the spectrum needs to be scaled proportionally to the flux density measured on the image pixel in terms of their coverage of the sky. Since the pixel covers a different (much smaller) area in the sky than the aperture from where the spectrograph collects the light, it is necessary to scale the flux density proportionally to the ratio of their coverages. Without this step, it would not be a flux density per area, it would be only an absolute flux and thus not comparable directly to the flux density coming from the image pixel.

Chapter 5

Sequential HiSS-Cube architecture

This chapter describes the part of the architecture that is specific to the sequential implementation of the HiSS-Cube.

5.1 Sequential HiSS-Cube File structure design

The HiSS-Cube File structure is depicted in Fig. 5.1. This data model is optimized for sequential implementation of the HiSS-Cube and while it is more efficient in the semi-sparse data combination on one processor, it becomes very inefficient if used for both database construction and querying in parallel. We describe a revised data model for parallel implementation of HiSS-Cube in Sec. 6.1. The sequential HiSS-Cube File data model is still interesting though, as it is very efficient for data visualization and storage due to more efficient linking of spectra and images and better support for compression of the HDF5 library in the sequential mode. Therefore, it can be used for smaller datasets where the *Scalability* requirement is not crucial. HiSS-Cube is configurable to work in either sequential or parallel mode.

One design option for the HiSS-Cube File data model was to merge all spectra into one dataset and images into one dataset, for each resolution respectively. However, mapping the original metadata to dataset regions representing the original images or to spectra is more complicated. Also, the HiSS-Cube File is harder to read interactively in existing HDF5 readers. On top of that, such merged datasets are harder to index and the read performance can be better optimized for the hierarchical index used for the individual datasets. The merged datasets are needed only for the parallel implementation, which we show further down in the Sec. 6.3. Having a hierarchical index and images with spectra as individual datasets provides better insert/update performance, even though that is not our primary goal.

Another considered option was to leave the data as external FITS files while using the HDF5 only for indexing and linking these files together and for storing only dense datasets in the HDF5 (slices of the cube). However, there is a complication in that the HDF5 region references cannot be used for external datasets, only internal ones. This is a severe

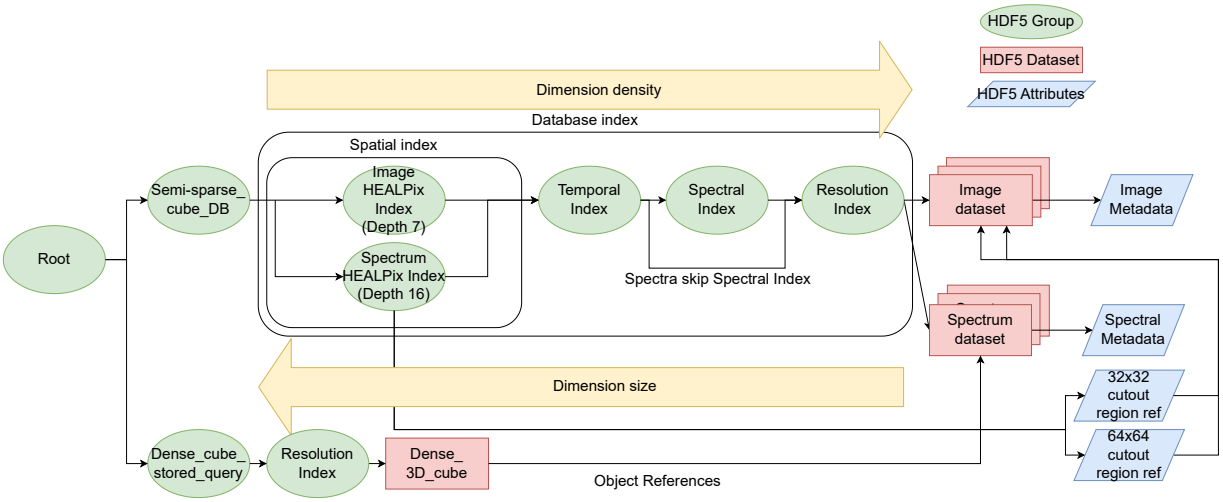

Figure 5.1: The first version of HiSS-Cube File structure optimized for sequential processing. The root group decomposes into two data structures - the *Semi-sparse_ cube_ DB* that contains original FITS files converted and preprocessed to HDF5 datasets, and the *Dense_ cube_ stored_ query* group that contains results of database queries in a storage optimized for sequential access. The semi-sparse data is indexed by the hierarchical *Database index* composed of *Spatial*, *Temporal*, *Spectral*, and *Resolution* indexes. Tree leaves beneath Database index are *Image Data* and *Spectral Data* Datasets, one for each original imported image or spectrum FITS file.

drawback because if we would like to mimic the region mapping on external files, we would for example end up with a tiny file for every image region and that would lead to inefficient access times.

The best approach in any case was to produce an indexing structure within the HiSS-Cube File that is used as a database view that links the spectra to their relevant image regions. The actual combination of the data then happens within a database query that uses this database view and creates slices of the combined data and might aggregate them along any axis. These stored query results would contain only data that is interesting for a particular research purpose. We also wanted to keep everything within one portable container of the HiSS-Cube File that will be self-describing (including all the data and metadata), so it can be processed independently by any other software that knows how to handle generic HDF5 files.

In our considerations of the architecture, we also took inspiration from the *Sparse chunks implementation* [37] approach. This paper includes comparisons of different design options for implementing semi-sparse data in HDF5. We went with the strong design option of reusing B-trees that are pervasive in the HDF5 library. However, contrary to Mainzer's suggestion, we did not use the B-trees to store all of the non-null values because it would be highly inefficient for dense parts of the data. Since B-trees are used within HDF5 for tracking each of the previously mentioned entities (HDF5 groups, datasets, or attributes),

we chose to use their functionality for indexing and organizing the data by a hierarchical group structure. Their usage for dataset chunking access is a convenient bonus that will be also leveraged as explained in Section 5.1.1. This results in a very good performance for the semi-sparse data combination but makes the parallelization much harder as described in Sec. 8.7.

One of our main contributions on top of HDF5 is a new indexing mechanism that HDF5 does not support by design. The reason why HDF5 does not support indexing is that an index is always domain-specific. This is especially the case when indexing physical coordinates, such as spherical coordinates (right ascension and declination for astronomy or longitude and latitude for Earth science). If HDF5 would include domain-specific implementation it would make HDF5 less flexible. There are libraries on top of HDF5 that implement indexing, such as PyTables, but these are hard to extend by a complex index, such as this spherical tesselation index, and do not apply well to multi-dimensional data.

The index is implemented as a group structure organizing the original FITS files that are stored as datasets within the HDF5 file, see Fig. 5.1. The preprocessed content of the original FITS file is stored as a dataset (1 *Image dataset* or *Spectrum dataset* dataset per 1 FITS file). These datasets form leaves of the B-trees that represent the hierarchical group structure. The flexibility of the groups allows also to use HDF5 Object or region references on the leaf nodes thus keeping the index structure apart from the data. We also needed to split both images and spectra into multiple groups, because there are millions of them. Inserting hundreds of thousands of datasets into one group causes the B-trees to overflow frequently and the tree splitting can become costly. This problem is solved by keeping the amount of elements in one group low, which can be done by the hierarchical index structure. Most impactful is the spatial index, as shown in Section 5.1.1.

The HDF5 offers two options for storing datasets: 1) **Contiguous storage** that stores the dataset as a contiguous array on disk. 2) **Chunked storage** that splits the dataset into regular slices of the same size. These chunks are in turn stored each as a contiguous array on disk and compression can be applied to individual chunks. This is a big strength of HDF5 because it enables us to read only relevant data before decompression and we used chunked storage for all bigger datasets.

The complete structure depicted in Fig. 5.1 is described below:

- *Semi-sparse_ cube_ DB* - It is an HDF5 group that contains the database index, image and spectral data, and their metadata. The *Image datasets* and *Spectra datasets* contain the original data from the image and spectrum FITS files that went through the preprocessing as described in Section 4.1.1. Since we combine spectra with images, we also interlink them with only relevant image data via HDF5 region references, which we describe in detail in Section 5.1.2. The *Semi-sparse_ cube_ DB* can be used for efficient visualization of the non-combined data as well. It also allows to read the relevant image region for each spectrum in an optimum number of disk I/O operations. Therefore, database querying and data combination is much faster. The *Image datasets* have three dimensions of size 2048x1489x2 where 2048x1489 is the size of

the original SDSS image and in the 3rd dimension, the mean and sigma values are stored (the original FITS images contain only the mean values and the sigma represents the uncertainty, see Def. 2.2.4). The *Spectrum datasets* have two dimensions of size 4620x2 where 4620 is the original resolution of the spectrum and in the 2nd dimension, the mean and sigma values are stored. Both *Image datasets* and *Spectrum datasets* are created for each lower resolution, respectively in their *Resolution Index* groups. Having each spectrum and image as an individual dataset also allows for different sizes of both to be stored in the database, in case heterogeneous data needs to be analyzed and combined.

- *Dense_ cube_ stored_ query* - It is a group that contains results of any database query run on the *Semi-sparse_ cube_ DB*. In the database language, these datasets are materialized views of the *Image* and *Spectrum* datasets. They typically contain only relatively small slices of the semi-sparse data and are by design optimized for bulk read access needed for further information extraction, for example, by machine learning algorithms.

5.1.1 Multidimensional indexing in HDF5

We were inspired by the HEALPix idea used in HiPS and we used it for indexing spatial dimensions. We have also good experience with using the HEALPix for unsupervised machine learning parallelization [42]. We have applied the HEALPix + HiPS idea to multidimensional data. Supporting the time and wavelength axes could also be solved by STMOC (Space-Time-Multi-Order-Coverage) and STEMOC (Space-Time-Energy-Multi-Order-Coverage) but only for coverage and not for data access. These were not yet part of the MOC Recommendation [15] but were already discussed here[1].

We present a hierarchical indexing structure on top of our data, implemented as native B-trees used for HDF5 group entities. This is the *Database index* in Fig. 5.1.

The composite index is a well-established technique used in relational databases to index data by multiple dimensions at once. We implemented the composite index as a hierarchical group structure. We can also leverage the feature of HDF5 to define the sorting order arbitrarily for each group. That is because HDF5 enables us to either sort the children of a group alphabetically or in the order in which the elements were created within that group. This gives the ability to keep the content human-readable, for example, the resolutions for individual MIP maps can be stored as strings, e.g., "2048x1489", "1024x744", "512x372", and still be sortable.

The most efficient structure of the index proved to be: Place more sparse and bigger dimensions in terms of distinct coordinates first. In our case, this results in the hierarchical index order of *Spatial >Temporal >Spectral >Resolution* coordinates. Apart from the visualization, this index layout is also optimized for the most common database query type: "Select" all images or spectra for a given spatial region on the sky, in a given time period, for a given resolution.

[1] `https://www.ivoa.net/documents/stmoc/index.html`

5.1.2 Region referencing in HDF5

We needed to address the sparsity in our data without aggregating it to the Contiguous HDF5 dataset[2]. The aggregated data structure also makes it harder to support visualization and interactive analysis of the data efficiently. For the use case of spectra and images, we wanted to combine image regions of pre-defined size with the spectra, since photometry measurements are only relevant to the spectral ones in a given spatial region of the sky from where the spectrograph collects the light. The relevancy will also degrade if the photometric measurement is far away in time, but the spatial condition is more constraining. The region references are implemented as hard links within an HDF5 file, containing the direct "pointer" (an offset) to the physical location on the disk. Since we use Chunked HDF5 datasets, de-referencing such a pointer is always followed exactly by one sequential read of the whole chunk.

Based on our results described in Chapter 8, this is the biggest strength of the HiSS-Cube sequential solution, since it enables an optimized read of all image regions relevant to the spectrum.

5.1.3 Attributes in HDF5

The attributes are created for all metadata from the original FITS file headers except for BITPIX, NAXIS, NAXISn attributes. This makes the HiSS-Cube File better human-readable through existing HDF5 clients that can visualize the attributes attached to each dataset or group natively. The excluded attributes are storage-specific and must be redefined in HDF5. We did not use the BITPIX (number of bits per pixel), because it does not make sense after compression. We decided to use the native C-style array ordering (also used in `NumPy`) instead of the Fortran-style, which is used in FITS standard[3]. This translates to different values in NAXIS and NAXISn attributes. It also makes it easier to manipulate the data, because the index will match the mapped `NumPy` multidimensional array that is directly provided by the `h5py` library API.

However, we ran into performance issues with the attributes for big data, especially in the `h5py` library, and therefore redesigned them in the Parallel version of the implementation as described within chapter 6

5.2 HiSS-Cube storage complexity

In this section, we discuss the storage complexity. Storage complexity means disk complexity because we consider big data that exceed the capacity of RAM or persistent memory (PM).

We describe only the impact of down-sampled resolutions and uncertainty on storage complexity since these have the biggest impact on HiSS-Cube File size. Stored query

[2] `https://support.hdfgroup.org/HDF5/doc/Advanced/Chunking/`

[3] `astropy` library also uses the Python/C-style array ordering

results can potentially also occupy a significant portion of the HiSS-Cube File but their size depends on the query completely and cannot be generalized. Indexing structures have also minimal impact since they are implemented as one-dimensional arrays that have linear storage complexity based on the input data.

5.2.1 Down-sampled resolutions

The storage complexity of data in down-sampled resolutions:

$$s_{res} = \sum_{i=1}^{n} \frac{s_{orig}}{2^{di}} \tag{5.1}$$

where s_{res} is the additional storage space needed by the precomputed down-sampled resolutions, s_{orig} is the original data size of images or spectra, n is the number of precomputed down-sampled resolutions and d is the number of dimensions of a dataset.

This results in the following series with the number of elements equal to the number of down-sampled resolutions we wish to keep.

For spectra ($d = 1$):

$$s_{res} = \frac{s_{orig}}{2} + \frac{s_{orig}}{4} + \frac{s_{orig}}{8} + \frac{s_{orig}}{16} + \cdots \tag{5.2}$$

For images ($d = 2$):

$$\frac{s_{orig}}{4} + \frac{s_{orig}}{16} + \frac{s_{orig}}{64} + \frac{s_{orig}}{256} + \cdots \tag{5.3}$$

Therefore, in the case of spectra, all down-sampled resolutions can increase the storage complexity by at most 100%, and in the case of images by at most 50%. However, for the example of our images which are 2048x1489 pixels, it does not make sense to precompute more than 4-5 down-sampled resolutions because too much information is lost. For spectra, we also keep only 4 down-sampled resolutions.

5.2.2 Uncertainties

Uncertainties always double the size of the data in each respective resolution regardless of the number of dimensions. Instead of storing one scalar number, two parameters are stored in the case of Gaussian distribution (mean and sigma). Probability distributions with a higher number of parameters will have proportionally higher storage complexity requirements.

Chapter 6

Parallel HiSS-Cube architecture

The HiSS-Cube architecture for the parallel implementation is the same as depicted in Fig. 4.1. However, the *Database construction* and *Database queries & Storing query results* need to be fully parallel. We still import FITS images and spectra into the HiSS-Cube File HDF5 format but we have optimized its structure for parallel access where possible. The data can still be efficiently accessed in the HiSS-Cube File for both data visualization and machine access.

6.1 Parallel HiSS-Cube File structure design

The second version of the indexing mechanism in HiSS-Cube enables parallel construction of the database as well as fast parallel queries on the database. The first tree-based version of the index proved to be inefficient and unscalable due to having several hundred million groups per one HiSS-Cube File, and the HDF5 library is not optimized for storing that many elements. We frequently ran into inefficiencies due to frequent B-tree overflows and subsequent splits, as well as severe degradation of performance caused by `h5py` overhead. We first implemented the index construction in C using directly the HDF5 C library to eliminate the `h5py` overhead. This resulted in the index construction scaling linearly with the amount of input data and the construction was significantly faster than in the case of `h5py`.

However, the C implementation of the database index construction still had one major design flaw - it could not be easily parallelized because of the hierarchical structure of groups within the HiSS-Cube File. Whenever the structure of a HiSS-Cube File is changed, which happens in the index construction all the time, it needs to be done as an MPI collective write operation [45]. A collective operation has to be done by all the writer processes so that each of them participates in every change of the structure. In contrast, independent MPI write operations can be executed by each process independently, without the need for inter-process communication or synchronization. Process synchronization for collective operations always slowed down a parallel construction of the hierarchical HiSS-

Cube File database index. The detailed performance measurements that support this fact are described in Sec. 8.7.

Therefore, we redesigned the HiSS-Cube File structure as shown in Fig. 6.1. The structure is similar to the previous version, but the database composite index is implemented as a single HDF5 dataset. For images, we constructed *Image_DB_index* (see Fig. 6.1), which represents a composite index of spatial, spectral, and temporal coordinates. For spectra, we constructed *Spectrum_DB_index*, which represents a composite index of spatial and temporal coordinates. The spectral index is missing for spectra because all of them have the same spectral dimension; therefore, the index does not add any information. The down-sampled resolutions of the spectra and images are organized into *Resolution_index* groups. This makes it much easier to parallelize thanks to the index itself being a multi-dimensional array rather than a tree.

Definition 6.1.1. Given multi-dimensional shape $[N, ...]$, a **stacked HDF5 dataset** is defined as a dataset with N rows of multi-dimensional arrays of the same shape [...].

Having one such stacked dataset has an advantage over having N datasets of the same shape [...] for data parallelism purposes, as described further in detail in Sec. 6.2. Image data and spectral data are stacked into *Stacked_images* and *Stacked_spectra* datasets, respectively, one for each resolution. One "row" within the *Stacked_images* dataset corresponds to one original FITS image but is enriched by additional preprocessed dimensions. The same applies to the *Stacked_spectra* dataset.

The original metadata from the FITS image and spectrum files are kept within their respective *Images_metadata_cache* and *Spectra_metadata_cache*. The *Stacked_images*, *Image_DB_index*, *Images_metadata*, and *Images_metadata_cache* datasets have all the same implicit array indexing. The same applies for the spectra datasets (*Stacked_spectra*, *Spectrum_DB_index*, *Spectra_metadata*, *Image_region_references* and *Spectra_metadata_cache*). Therefore, finding the metadata for the relevant image is trivial (accessing the same offset), and the data structures are kept dense for storage efficiency and easy parallel splitting of workloads. To maintain flexibility, we used JSON-like storage within metadata datasets.

This makes the number of entries in the HiSS-Cube File dependent only on the number of down-sampled resolutions, instead of on the number of imported images and spectra, or on the size of their database indexes. This allowed all phases of *Parallel database construction* (see Fig. 4.1 and Fig. 6.2) to be fully parallelized, as described in Sec. 6.2.

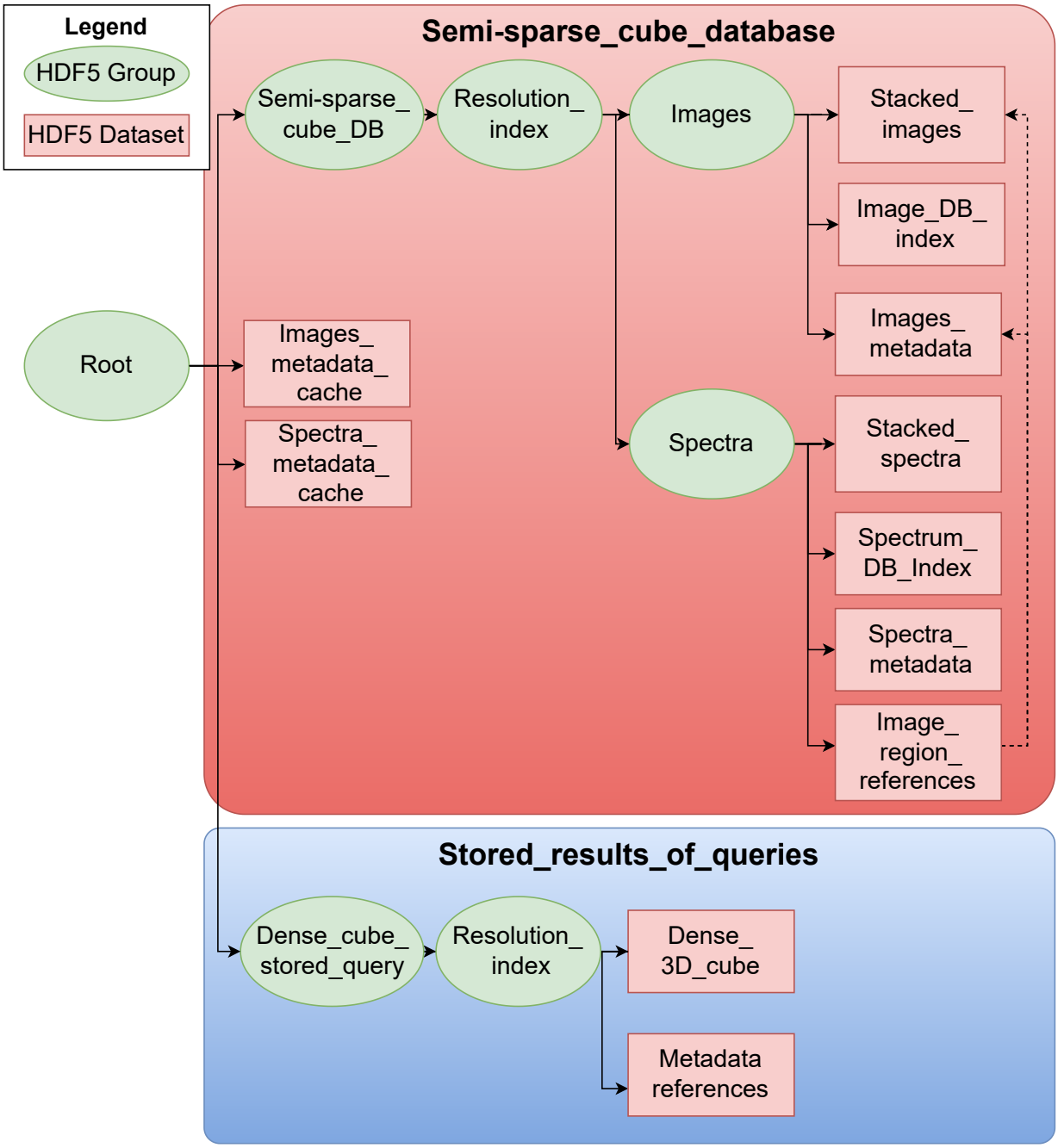

Figure 6.1: The second version of HiSS-Cube File structure optimized for parallel processing. This whole structure is encapsulated within a single `.h5` file consisting of two parts, similarly to the previous file structure seen in Fig. 5.1: (1) The *Semi-sparse_ cube_ database* that stores the enriched FITS images and spectra and contains their database indexes for fast searching. (2) The *Stored_ results_ of_ queries* part that can store results of queries run on the *Semi-sparse_ cube_ database* in all resolutions. The *Dense_ cube_ stored_ query* group contains the results of all stored queries. An example of such a stored query result is the *Dense_ 3D_ cube* dataset which aggregates the 4D semi-sparse data into a 3D cube.

6.2 Parallel architecture

The HiSS-Cube *Parallel database construction* part (see Fig. 6.2) consists of five consecutive fully parallel phases. MPI was used for parallelization with its `mpi4py Python` wrapper. We did not use threads because HiSS-Cube processing is I/O-bound, and the processes are more efficient at handling it. This means that all process synchronizations occur at the barriers at the end of each phase. Inside the phases, all workers run independently, without any additional inter-process communication.

We distinguish between a **serial mode**, where only one process may read from or write to the HiSS-Cube File, and a **parallel mode**, in which all processes can read from or write to it.

The changes in the HiSS-Cube File structure, such as pre-allocation of new HDF5 datasets, creation of HDF5 groups, or writing HDF5 attributes to those groups, can in principle be done in parallel by HDF5 collective operations of the *Parallel database construction*. However, we have experienced that within each phase, it is more efficient to do it with a single writer in serial mode of the *Parallel database construction* and then to reopen the HiSS-Cube File in parallel mode by all the processes.

This causes complications for some of the HDF5 Virtual File Drivers (VFDs)[1], such as the Subfiling VFD, which always needs to open the HiSS-Cube File in parallel mode with the MPIO file driver. This can be overcome by creating an MPI sub-communicator for a single writer who writes alone without additional inter-process synchronization.

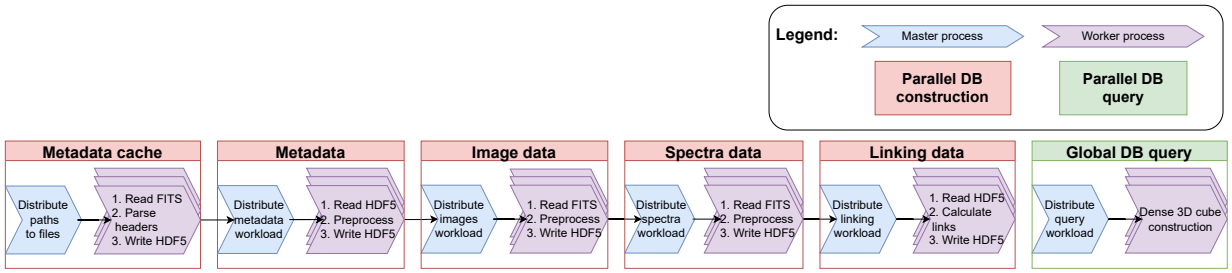

Figure 6.2: HiSS-Cube implementation of both the *Parallel database construction* and the *Parallel DB Query*. The *Parallel database construction* is further split into 5 consecutive phases. In these, workers either read data from the HDF5 HiSS-Cube File or from the FITS input files, then do the computation, and then write the result to the HDF5 HiSS-Cube File. The *Global database query* is more complicated but also uses the Master/Worker architecture. The *Global database query* is one example of possible *Parallel DB Query* on the HiSS-Cube File *Semi-sparse_ cube_ database*.

The phases involved in *Parallel database construction* are listed in Fig. 6.2. We used the Master/worker MPI architecture, where workers always use *independent MPI I/O* [45] operations on the HiSS-Cube File. The Master distributes the workload uniformly in

[1] `https://www.hdfgroup.org/wp-content/uploads/2021/10/HDF5-VFD-Plugins-HUG.pdf`

batches to maintain the local processing time of each batch at approximately 10 seconds. This results in a tradeoff between the number of MPI messages sent and the time spent at the end of each phase, where processes wait on the barrier for other processes to finish their batches. We verified that the Master process was not a bottleneck for tens of thousands of workers. If a single Master becomes a bottleneck, replacing it with multiple ones is trivial.

6.2.1 Parallel compression

HDF5 and `h5py` both support parallel compression, however, it is supported only for collective write and read operations. The collective write operations are typically slower than independent ones for a bigger volume of data and a small total number of write operations. These kinds of write operations are the bottleneck for HiSS-Cube write performance. Since our goal was maximum write and read performance of big portions of data, we did not implement parallel compression in HiSS-Cube. This would change if the feedback from the scientific community would require saving storage space at the cost of speed of the database construction or querying.

6.3 Parallel database construction

In this section, we describe how the individual phases process the data and write it into the HiSS-Cube File (see Fig. 6.1 and Fig. 6.2).

1. *Metadata cache* phase: The Master opens the HiSS-Cube File in a serial mode and allocates the *Images_ metadata_ cache* and *Spectra_ metadata_ cache* datasets. The sizes of these datasets are statically defined by configuration based on the user's requirement for the number of imported images and spectra. Subsequently, the Master closes the HiSS-Cube File and all processes reopen it in parallel mode.

   The Master recursively traverses the directories with the images and spectra FITS input files to obtain their paths. It continuously distributes these paths in batches to workers who open these files independently. A worker parses the header of each spectrum or image FITS file and writes the metadata it contains in a JSON-like format to the *Images_ metadata_ cache* and *Spectra_ metadata_ cache* datasets, respectively.

   In the case of dynamic allocation of datasets, the Master would need to traverse the whole directory structure of the input FITS files to count them, which is time-consuming. This would need to be done before allocating the *Images_ metadata_ cache* and *Spectra_ metadata_ cache* datasets and blocking the workers until it happens. Therefore, we used the static allocation.

   After both datasets are allocated, the Master can distribute the paths to the workers continuously as he traverses the directory structure. The directory traversal by the Master could be a bottleneck if scaled to hundreds of thousands of workers, but this phase does not take significant time for the overall *Parallel database construction*; therefore, we did not optimize it further.

2. *Metadata* phase: The Master opens the HiSS-Cube File in serial mode and allocates the *Image_DB_index*, *Images_metadata*, *Spectrum_DB_index* and *Spectra_metadata* datasets. It knows their exact sizes from the number of imported images and spectra metadata in the previous phase. Subsequently, all processes open the HiSS-Cube File in parallel mode. The Master then takes the count of imported images (which is stored as an attribute within the HiSS-Cube File), constructs a list of row indexes for both the *Images_metadata_cache* and *Spectra_metadata_cache* datasets, and distributes them to the workers.

   The workers read the metadata stored in *Images_metadata_cache* or *Spectra_metadata_cache* in parallel, recalculate it for each down-sampled resolution, and store it within the *Images_metadata* and *Spectra_metadata*, respectively, for each down-sampled resolution. This involves recalculating the pixel sizes or spatial coordinate projections for the down-sampled resolutions to keep them scientifically accurate.

   The workers also write the *Image_DB_index* or *Spectrum_DB_index* dataset entry for the given image or spectrum in the same step because the information regarding their coordinates is also stored within the parsed metadata in the *Images_metadata_cache* or *Spectra_metadata_cache*, respectively. As mentioned in Sec. 6.1, these database indexes serve the database queries that search by spatial, spectral, or temporal coordinates. The *Image_DB_index* and *Spectrum_DB_index* datasets are written in parallel, but they must be sorted afterward. This is called clustering of the index in the database community.

   Currently, sorting is performed in-memory by the Master at the end of this phase in serial mode, but in principle, it can be parallelized as well. This is the main trade-off in the design of the HiSS-Cube File structure. Having a database index implemented as an array makes parallel creation of the index efficient, but every insert or update would require resorting of the index. However, this is not an issue because HiSS-Cube is optimized for database creation once per data release, and then the database will be read-only. Moreover, as measured in the performance (see Sec. 9.3), the index creation operations can process tens to hundreds of thousands of images or spectra per second on our hardware, so even recreation of the index is very fast. Therefore, the ability to create the index in parallel also outweighs the inefficiency of update and insert operations.

3. *Image data* phase: The Master allocates the *Stacked_images* dataset in serial mode. After that, all the processes open the HiSS-Cube File in parallel mode. The Master then distributes indexes of rows from the *Images_metadata_cache* dataset, similarly to the *Metadata* phase.

   The workers read the paths to the images from *Images_metadata_cache* and use them to open the image FITS files in parallel. Subsequently, the workers read the data from these files independently, preprocess it, and write the result to the pre-allocated *Stacked_images* datasets in their respective *Resolution_index* groups. This

phase is also computationally expensive because it involves many activities, such as error reconstruction or creating images in down-sampled resolutions (see Fig. 8.3).

4. *Spectra data* phase does the same for spectra.

5. *Linking data* phase: The Master allocates the *Image_region_references* dataset in the serial mode. Subsequently, all processes open the HiSS-Cube File in the parallel mode. The Master then distributes the indexes of the rows of the *Stacked_spectra* to the workers.

   Workers query the *Image_DB_index* dataset to find all the image regions that overlap the spectrum coordinates obtained from the *Spectra_metadata* dataset. For these overlaps, workers generate region references and store them in the *Image_region_references* dataset for each resolution. These references are then used to link a particular spectrum with its overlapping image region in the *Stacked_images* and its relevant metadata within the *Images_metadata* dataset, again in the corresponding resolution.

   Owing to the need for independent writes, custom-structured datatypes must have been implemented to store these region references. HDF5 region references[2] do not have support for independent parallel writes, because they are variable-length datatypes.

6.4 Global database query - Combining of all spectra with all images

To prove the scalability of our HiSS-Cube implementation, let us describe an example of a database query that can run on top of the *Semi-sparse_cube_database* and store its results in *Stored_results_of_queries* for further efficient access. We used this query to benchmark the performance of the HiSS-Cube database within HDF5 and its behavior with many small random reads and writes in a realistic scientific scenario [55].

The *Global database query* was also designed to be fully parallel owing to the convenient HDF5 structure of the HiSS-Cube File.

During the *Global database query* workers:

1. Get the spectra for each assigned object from the *Stacked_spectra* and aggregate them along the time axis, resulting in one spectrum for each observed object.

2. Get all the overlapping image regions covering this object from the *Stacked_images* and again aggregate these along the time axis.

This is performed for each down-sampled resolution of the images and spectra. This results in pairs of one spectrum and one image region for each observed object and resolution.

[2] `https://docs.h5py.org/en/stable/refs.html`

Workers store the results of this query in the *Dense_3D_cube* datasets inside the *Dense_cube_stored_query* group. The references to the metadata of the original spectra and images are stored within the *Metadata_references* dataset in the same *Resolution_index* group as the *Dense_3D_cube* dataset at the same resolution.

Let us describe the individual steps of the *Global database query* in more detail. At the beginning, the Master allocates the *Dense_3D_cube* dataset and the *Metadata_references* dataset in serial mode. The size is estimated from individual spectra locations obtained from the *Spectrum_DB_index* dataset. Subsequently, all processes open the HiSS-Cube File in parallel mode. The Master distributes the indexes of the rows from the *Stacked_spectra* (which are the same as the indexes of their corresponding *Image_region_references*).

The workers then read all assigned spectra from the *Stacked_spectra* and aggregate them along the time axis. They also read all assigned overlapping image regions from the *Stacked_images* and aggregate them along the time axis. Subsequently, they write this aggregated result to the *Dense_3D_cube* dataset in its respective *Resolution_index* group. They also write references to the *Spectra_metadata_cache* and *Images_metadata_cache* into the *Metadata_references* dataset.

Because not every spectrum has an overlapping image, there would be vacancies in the *Dense_3D_cube* and *Metadata_references* datasets, making them rather sparse. Therefore, we added one subphase that shrinks the *Dense_3D_cube* dataset using a parallel prefix sum. The Master collects the number of **complete** *Dense_3D_cube* rows written by each worker.

Definition 6.4.1. Complete *Dense_3D_cube* row is a 3D cube, where the spectrum has at least one overlapping image region in each of the five colors so that a dense 3D cube can be constructed without missing data.

From these collected counts, new offsets are calculated, using the prefix sum, where these workers should write the complete *Dense_3D_cube* rows. Subsequently, it creates a new *Dense_3D_cube* dataset of the desired size. Subsequently, the workers write all the data in parallel mode to this new dataset to the offsets distributed by the Master. After all the workers are finished, the Master deletes the old *Dense_3D_cube* dataset and renames the new one in the serial mode.

The results of the *Global database query* phase are pairs of spectra with their matching image regions for all observed objects, for which we have both a spectrum in the catalog of spectra and a complete image region in all five colors. This is the *Dense_3D_cube* dataset with dimensions (right ascension, declination, and wavelength), and the time dimension is aggregated. An example of visualization of it using the `H5Web` tool is shown in Fig. 6.3. We used the `Nexus` data format [29] annotation for *Dense_3D_cube*, making this visualization available in one click on the `H5Web` client.

These stored query results can be further used for machine learning algorithms, such as classification or clustering of the types of objects observed, not only based on images or spectra individually but also based on the **combined** information. Another interesting

application that we are currently working on is the estimation of star formation rates [10] from the combined spectra and images.

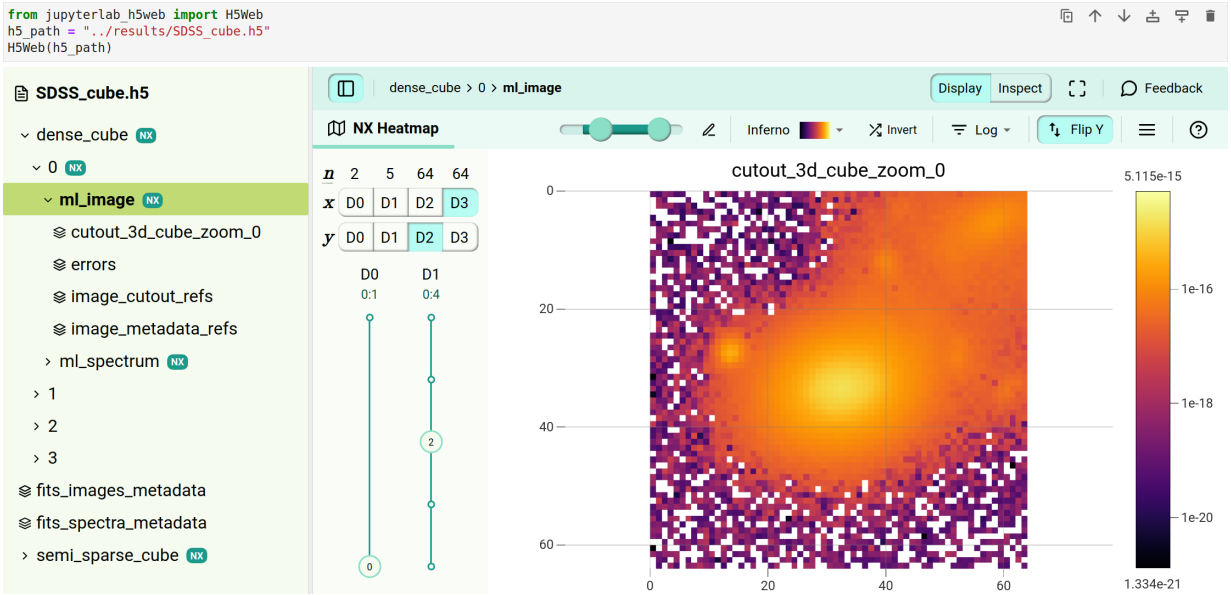

Figure 6.3: Demonstration of the `H5Web` tool for visualization of big data stored within HDF5. This figure shows a screenshot of a `jupyter lab` visualizing one slice of the *Dense_3D_cube* dataset within the HiSS-Cube File (slice is an image of a galaxy in one of the five filters).

Chapter 7

Implementation

This chapter describes the most interesting parts of the HiSS-Cube implementation. It is not a complete documentation but rather highlights the key decisions that were made during the implementation.

The technologies we used were already mentioned in Chapter 4. In addition to those, we used the `astropy` and `fitsio` libraries for handling the reading of FITS files, pure `NumPy` for the preprocessing part, and `healpy` + `h5py` libraries for manipulating the HDF5 file itself. For data export to VOTable, we used the `astropy` library as well and the machine access interface is mapped by the `h5py` library directly to a `NumPy` array. By this, HiSS-Cube fulfills the requirement *Modularity, and Flexibility*, resulting in a framework for any dimensionally multi-modal data combination, rather than just a specialized one-purpose software.

As already mentioned, no mature analytical software exists for the semi-sparse data. Because of the absence of specialized software, we used tools such as `H5Web` for data analysis and visualization, in combination with `Python` utilities working with the `h5py` library.

7.1 Generic HiSS-Cube implementation

This section describes the generic implementation parts that are used independently of whether the application is running in sequential or parallel mode.

7.1.1 Database construction

In this section, we show interesting parts of the preprocessing code, which is part of the database construction. The snippet below represents the most time-consuming part of the preprocessing - extracting the uncertainties from the calibrated images. The process is described in more detail in Section 2.2.4 - it reverts the calibration applied to the published FITS images almost precisely since of course, floating-point operations are not entirely reversible. However, the image pixels are compressed with a lossy compression preserving strictly 10 bits of the mantissa of the 32-bit floating-point numbers. This compression does

not reduce per-pixel accuracy by more than one percent[1]. This is potentially much more than the error introduced by multiple floating-point operations on every pixel of the image. Hereinafter, we refer to this lossy compression as *Float Compression*. When writing the preprocessed data into the HDF5 file, we apply also the *Bit shuffling* and GZIP compression using the HDF5 Filters functionality[2] that enables us to chain the compression methods easily.

The inputs to the code below are three parameters: `img` - the original 2048x1489 image read from the FITS data, `allsky` - 256x196 (usually) sky background image included within another extension of the FITS, `calib` - calibration vector used for the reduction of this image. This vector has only one dimension because of the way SDSS optical cameras work and can be safely expanded to the whole image as seen below. The contents of `gain` and `darkVariance` variables are described in Section 4.1.1.

```
#interpolating sky background from 256x196 image
simg = ndimage.map_coordinates(allsky, (grid_y, grid_a), order = 1,
    mode="nearest")
#calibration image constructed from calibration vector
cimg = np.tile(calib, (y_size, 1))
#reversing calibration to get the original data numbers
dn = img / cimg + simg
#calculating uncertainty (sigma) from data numbers
dn_err = np.sqrt(dn/gain + darkVariance)
#converting uncertainties to nanomaggies (sigma)
img_err= dn_err*cimg
```

The following snippet efficiently finds images that overlap a spectrum defined by the fiber position (`PLUG_RA` and `PLUG_DEC`). By using the HEALPix hierarchical structure (`hp.ang2pix`), the snippet constructs the path in the spatial index (`heal_path`) and finds the image locations on a disk directly, without walking the spatial index tree on the disk. This method is fast, most of the time is actually spent calculating the region bounds via the FITS WCS.

```
pixel_IDs = hp.ang2pix(
    hp.order2nside(np.arange(self.IMG_SPAT_INDEX_ORDER)),
    self.metadata["PLUG_RA"],
    self.metadata["PLUG_DEC"],
    nest=True,
    lonlat=True)
 heal_path = "/".join(str(pixel_ID) for pixel_ID in pixel_IDs)
heal_path_group = self.f[heal_path]
for time in heal_path_group:
    time_grp = heal_path_group[time]
    for band in time_grp:
```

[1] http://www.sdss3.org/dr8/software/idlutils_doc.php#FLOATCOMPRESS
[2] https://portal.hdfgroup.org/display/support/Filters

The FITS WCS parsing from the original FITS headers stored in HDF5 was very slow but we managed to speed it up cca ten times by not relying on the generic parser of the whole header, but rather by building the WCS programmatically from the absolute minimum of header attributes. Afterward, the region bounds are calculated as follows.

```
w = get_optimized_wcs(image_ds.attrs)
image_size = np.array((image_ds.attrs["NAXIS1"], image_ds.attrs["NAXIS2"]))
pixel_coords = np.array(skycoord_to_pixel(
    SkyCoord(ra=spectrum_fits_header["PLUG_RA"],
        dec=spectrum_fits_header["PLUG_DEC"], unit='deg'),
    w))
#we only take regions that are whole 64x64
if 0 <= pixel_coords[0] <= image_size[0] and 0 <= pixel_coords[1] <=
    image_size[1]:
    pixel_coords = (pixel_coords[0], pixel_coords[1])
    region_size = int(self.SPECTRAL_cutout_SIZE / (2 ** res_idx))
    cutout_bounds = np.array(
        (int(pixel_coords[0]) - (region_size / 2),
        int(pixel_coords[1]) - (region_size / 2)) ...
#getting the region reference from HDF5 dataset
region_ref = image_ds.regionref[cutout_bounds]
#saving to HDF5 attributes
spec_ds.attrs["image_cutouts"] = image_refs[spec_res_idx]
```

7.1.2 HDF5 file manipulation

The HDF5 manipulation is perfectly seamless for example for `astropy.wcs` module as seen on the 1st row in the code above. If we were not optimizing that part, we could have just `w = wcs.WCS(image_ds.attrs)` where `attrs` is the same Python dictionary-like object as a FITS header.

Another example of easy `NumPy` manipulation of the data is access to image regions. The `image_region` would be in our case a `NumPy` array of shape (64, 64, 2). In this array, 64x64 pixel images are stored, with mean and sigma values at each pixel. The dereferencing of region references has two steps. First to get the dataset `image_ds` and second to get the region of that dataset `image_region`.

```
image_dataset = hdf5_file[region_ref]
image_region = image_ds[region_ref]
```

7.1.3 VOTable export

Here we show how easy it is to convert `NumPy` arrays to VOTable [43] and send it by SAMP [61] protocol to Virtual Observatory tools. The `NumPy` array is passed to `astropy` just by accessing the `dense_cube` HDF5 dataset inside the opened `hdf5_file`. The empty `[()]` is there to pass the whole `NumPy` array, not just a smaller slice of it.

```
table = QTable(hdf5_file["dense_cube"][()],
    names=("HealPix ID", "RA", "DEC", "Time", "Wavelength", "Mean", "Sigma"),
    meta={'name': 'SDSS Cube'})
votable = from_table(table)
writeto(votable, output_path, tabledata_format="binary")
client = SAMPIntegratedClient()
client.connect()
params = {"url": urljoin('file:', os.path.abspath(output_path)), "name": "SDSS
    Cube"}
message = {"samp.mtype": message_type,
           "samp.params": params}
client.notify_all(message)
```

7.2 Sequential HiSS-Cube implementation

In this section, we show part of the application class/module architecture that enables easy changes and replacements of the SDSS-specific data processing, or even processing of images and spectra for different types of data.

Let us describe the overall architecture for *Database Construction*. As depicted in Fig. 7.1, we used the Builder design pattern where a Director defines the set of builders that should be used for the construction based on HiSS-Cube configuration and its `construct()` method calls in a sequence the `build()` method of each builder. As such, each builder is interchangeable with another builder that implements the `build()` method. Each builder builds a part of the HiSS-Cube File.

The `SingleSpectrumBuilder` and `SingleImageBuilder` are used in both serial and parallel builders and illustrate that they can be also replaced by a completely different builder that builds the same part of the HiSS-Cube File. Such a builder could be building different types of data than images or spectra.

There are also other builders for building arbitrary parts of the HiSS-Cube File which are not needed for the spectra and images combination, but rather the follow-up knowledge extraction by machine learning algorithms, such as the `SFRBuilder` which imports the star formation rates and matches them as labels for spectra already imported by the other builders.

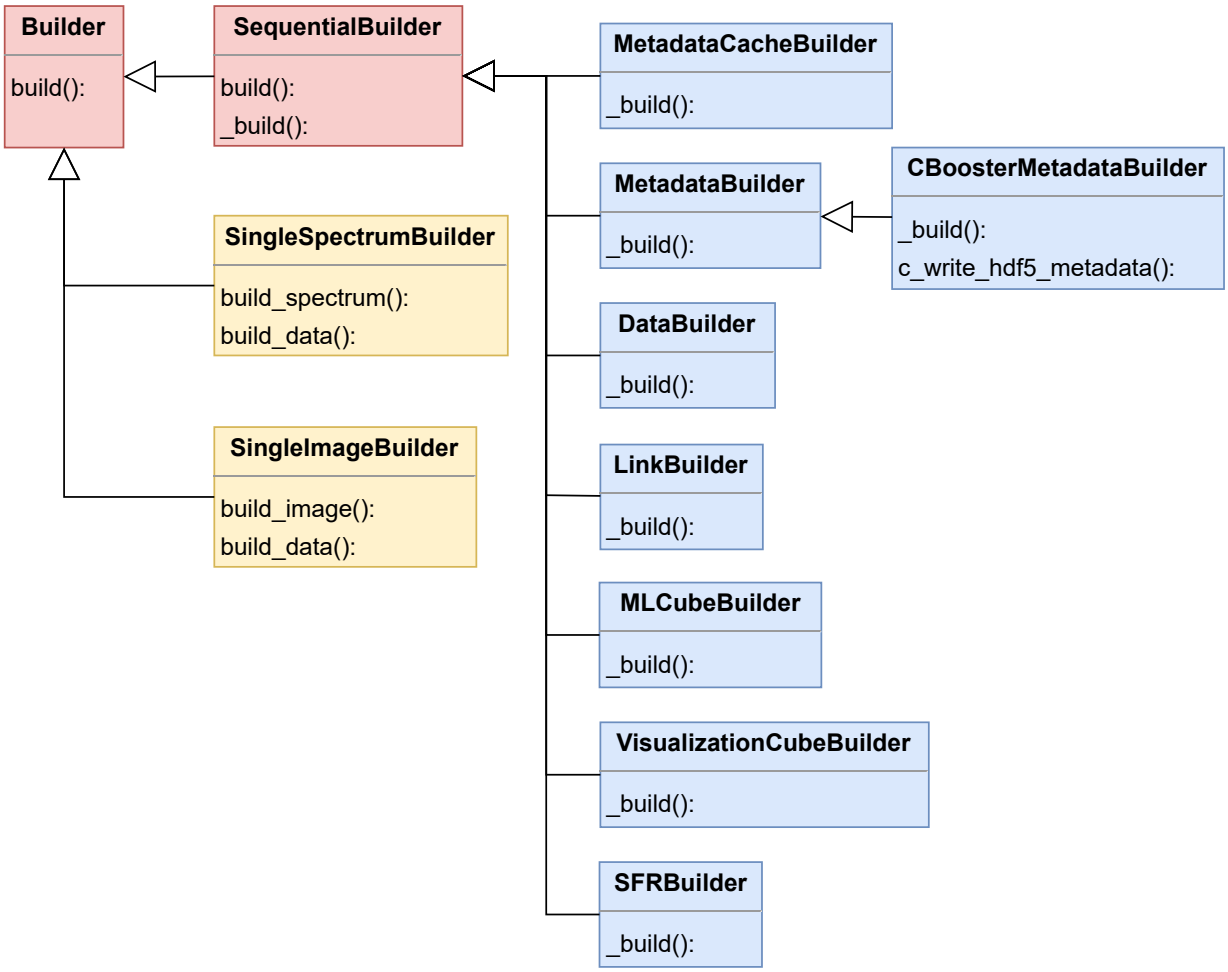

Figure 7.1: Class diagram of the sequential builders used within HiSS-Cube to construct the HiSS-Cube File. The red builders are abstract classes, the yellow ones are utility for building a single spectrum or image, usually used by other builders. The blue builders build parts of the HiSS-Cube File within their respective phases.

7.3 Parallel HiSS-Cube implementation

This section describes the parallel implementation parts of the HiSS-Cube framework. The builder architecture is depicted in Fig. 7.2.

The parallel versions of the builders do the same thing as their serial counterparts - import images and spectra, but they create different structures within the HiSS-Cube File, as described within Sec. 6.1. This illustrates that different strategies can be adopted for importing the same data in multiple ways. These parallel builders also reuse the serial builders where possible, further increasing the modularity of HiSS-Cube.

There are some builders that were not implemented for the parallel version, such as the `VisualizationCubeBuilder`, which experimented with exporting to VOTables and FITS files. This approach was deprecated for big data in the parallel version in favor of the

`H5Web` client with `Nexus` annotation for better flexibility.

Note that the variants of serial and parallel builders are completely exchangeable in the HiSS-Cube building process. There can also be multiple implementations of the same builder which were evaluated for their performance, such as *ParallelMWMRDataBuilder* (Multiple Writer Multiple Reader) and *ParallelSWMRDataBuilder* (Single Writer Multiple Reader). The SWMR data builder was deprecated since the MWMR data builder provided much better overall performance.

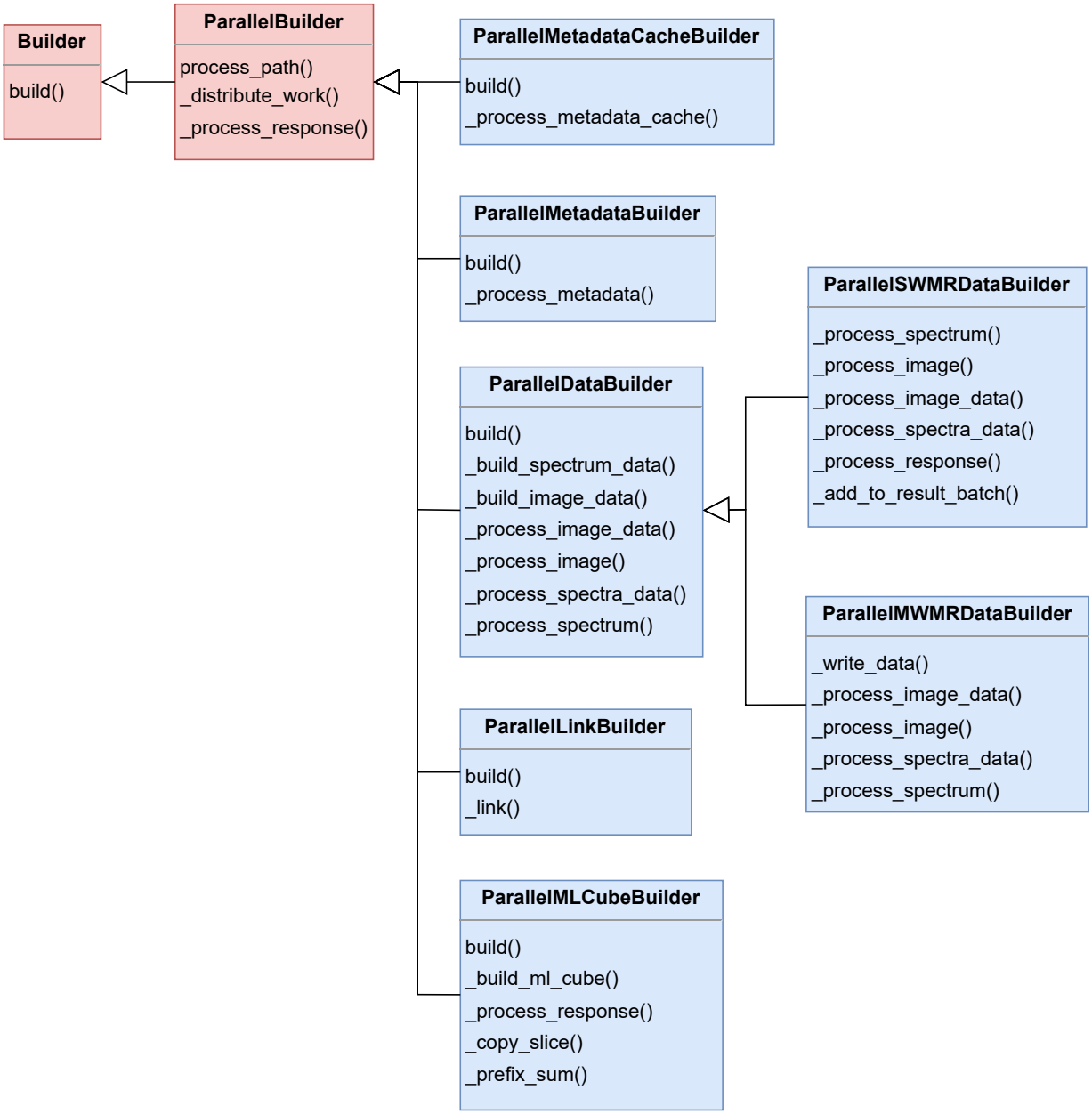

Figure 7.2: Class diagram of the parallel builders used within HiSS-Cube to construct the HiSS-Cube File. The red builders are abstract classes, and the blue builders build parts of the HiSS-Cube File within their respective phases. While the parallel builders are much more complex than their sequential variants (see Fig. 7.1), they are completely interchangeable because they implement the same Builder interface.

7.4 Modularity - HiSS-Cube extension examples

In this section we show how the modular architecture depicted in Figures 7.1 and 7.2 can be reused and extended for other types of data or data from different detectors.

In principle, all of the parallel versions of the data builders (such as *ParallelMWMRBuilder*) reuse their serial builder variants and only implement the additional layer of parallelization, so no modification to the Parallel builders should be needed.

7.4.1 Different detectors with the same type of data

When importing the same type of data (spectra and images) into HiSS-Cube, the metadata will likely be stored differently in the source data format (different naming convention, different structure). To alleviate that, parts of *MetadataBuilder* need to be extended and overridden. These parts are decoupled further into small parts within the single responsibility principle, so the developer can reuse the maximum of the already existing HiSS-Cube functionality.

If importing other data format than `FITS`, the *MetadataCacheBuilder* needs to be extended and overridden. If the metadata would have the same structure, no subsequent changes to *MetadataBuilder* would be needed. If the metadata would be described in key-value pairs, then essentially only this line of the code needs to be changed and the dictionary provided by a different reader than the `fitsio` FITS header parser:

```
        serialized_header = ujson.dumps(dict(fitsio.read_header(path)))
```

If the structure of metadata would be different, parts of *MetadataBuilder* would need to be extended and overridden as well to match these differences.

If the data were stored in a different form, the *SingleImageBuilder* and the *SingleSpectrumBuilder* would need to be extended and overridden to match those differences. The *SingleImageBuilder* has also decoupled parts within the *Photometry* module that serve as a library of photometry-related transformations and can be reused and extended easily. As we use dependency injection, a different module for the photometry can be injected at initialization time as well, as seen below:

```
class SingleImageBuilder(Builder):
    def __init__(self, config: Config, h5_connector: H5Connector,
                 image_processor: ImageProcessor,
                 photometry: Photometry, logger, image_path=None):
        super().__init__(config, h5_connector, logger)
        self.image_path = image_path
        self.image_processor = image_processor
        self.photometry = photometry
```

No further changes are needed, since when combining images and spectra, the builders for linking and combining the data will behave the same.

7.4.2 Increasing dimensionality of semi-sparse data

If we need to add another dimension, for example, a polarization to an image, we would need to extend and override the *MetadataBuilder* to count with this additional dimension when building the structure of the HiSS-Cube File and allocating the `Stacked_images` dataset (see Fig. 6.1).

Then, the *SingleImageBuilder* would need to be extended and overridden to produce the polarized data as another dimension of the `NumPy` array that it produces within the `get_multiple_resolution_image`, see code below.

```
def build_data(self, h5_connector, image_path, image_metadata_processor,
    fits_file_name, offset=0):
    metadata, data =
        self.photometry.get_multiple_resolution_image(image_path,
        self.config.IMG_ZOOM_CNT)
    res_grp_list = image_metadata_processor.get_resolution_groups(metadata,
        h5_connector)
    img_datasets = image_metadata_processor.write_datasets(res_grp_list,
        data, fits_file_name, offset)
    return img_datasets
```

Afterward, the rest of the builders will treat the polarization seamlessly as another dimension of the data and do not need to be changed.

If the polarization needs to be part of the data linking or combination, the *LinkBuilder* and the database query (such as *MLCubeBuilder*) need to be extended and overridden, respectively.

7.4.3 Combining data from more than two detectors

To combine the data from three or more detectors, the *MetadataCacheBuilder* and *MetadataBuilder* need to be extended and overridden to produce additional groups with the same structure as `Images` or `Spectra` in Fig. 6.1. The location of `Image_region_references` dataset defines the direction of linking of the data. If present, for example, in the `Spectra` group, it means the follow-up database query, such as *MLCubeBuilder*, will be finding the images relevant for the spectra and not the other way around. Generally, the direction of the link should be from smaller data to bigger data, but this does not need to be the case and can be also bidirectional in principle. Since the links are implemented and stored as `Python tuples` of the coordinates combined with the group reference to the dataset they are linked to, adding additional linking datasets is not expensive in terms of additional storage space consumed.

Afterward, the *LinkBuilder* and the database query (such as `MLCubeBuilder`) need to be extended and overridden to link the data correctly (linking is constructing the *Image_region_references* dataset) and use these links correctly when querying the data and combining it.

7.4.4 Extending data structure to SKA data

To add different types of data than spectra and images, such as a spectral data cube[3], the *MetadataBuilder* needs to be modified to allocate the additional dimension to the data, similar to polarization in the Sec. 7.4.2. For the SKA data cubes, this would result in 6D *Stacked_spectral_cubes* dataset (spatial in 2D, spectral, temporal, polarization, and uncertainty dimensions), similar to `Stacked_images` dataset in the `Images` group (see Fig. 6.1).

For the spectral data cube data, we would need to extend the *SingleImageBuilder* or create a completely new one called *SingleSpectralCubeBuilder* which will produce the 6D multi-dimensional array instead of the 4D in case of the *SingleImageBuilder*.

Then, the *LinkBuilder* would need to be extended and overridden to link the spectra (or images) correctly to their relevant regions within the *Stacked_spectral_cubes* dataset. In the case of linking spectra to the spectral data cube, the changes would be minimal.

The database query (such as *MLCubeBuilder*) would also need to be different to be able to combine the additional dimension. However, in the case of combination of spectra with the spectral data cubes coming from SKA, the database query would be very similar: For each spectrum, return its relevant region within the spectral data cube and combine the data. Therefore, extending and overriding the functionality of the *MLCubeBuilder* database query would be rather easy.

7.5 Data integrity

We implemented algorithms to detect two kinds of errors in the HiSS-Cube framework:

1. Data corruption or inconsistency in the input data. This means that either the data is corrupted, or the metadata is corrupted or inconsistent. For example, the astrometric computation from the WCS metadata cannot be performed because it was corrupted either by machine faults or human errors. This causes the worker is then unable to process the data.

2. Worker crash: A worker process crashes and therefore his current batch is lost.

In case of error (1), the worker recovers and continues processing the rest of its batch, which results in a blank spot in the semi-sparse database. At the moment we are not "shrinking" the dataset to remove these blank spots because they are rare. In case of error (2), HiSS-Cube terminates the whole phase because of the risk that HiSS-Cube File would be corrupted (HDF5 does not have a journal or other mechanisms for restoring corrupted parts of an HDF5 file). This data corruption is corrected by re-running the respective phase which reallocates all the datasets belonging to it and thus cleans any inconsistencies. We have not considered implementing journal-based recovery because re-running the whole phase is fast enough.

[3]Spectral data cube is sometimes called image data cube

Chapter 8

HiSS-Cube sequential performance & functional evaluation

8.1 Hardware

We ran the sequential performance tests on a laptop with 8-core AMD Ryzen 7 4800HS processor, 40 GB RAM 3200MHz DDR4, and NVMe SSD Western Digital PC SN530.

This fast SSD is completely sufficient for benchmarking the HiSS-Cube for sequential performance, especially compared to slower HDDs frequently used for HPC big data storage. On an SSD, the test results are not impaired by the randomness of disk seek times and thus provide a more reliable picture of the HiSS-Cube efficiency in the number of I/O operations. All our tests were run ten times and the results were averaged across all runs to minimize measurement errors. We also flushed the memory caches (`PageCache`, `dentries`, and `inodes`) before every test. All of the experiments were performed on a single core.

8.2 Test data

To be able to run the benchmarks in a reasonable time but also on big enough data to be reliable, we chose two galaxies close to each other in the sky in the **Stripe 82** field (a particularly dense region within the SDSS data), counting 11 spectra and 185 images overlapping these spectra. This is one of the densest areas of the SDSS survey, that is, the highest amount of images observed at different times which overlap each other on the sky. The relevant overlaps are shown in Fig. 8.1 with other rotations and slices of the data in Fig. 8.2. The size of this SDSS decompressed dataset is 2.2GB, see more details in Section 8.5. Hereafter, we refer to this test data as **Galaxy2**.

8.3 Visualization of the Galaxy2 data cube

In this section, we show the HiSS-Cube visualization results on Galaxy2 data in TOPCAT [60]. This is the most popular tool for the visualization of tabular data in astronomy and by using it we have shown that our data structure can be used for explorative analysis of the data with existing clients and IVOA standards[1]. Even though the image regions are only 64x64 pixels in size, there are still cca 4 million 4-dimensional voxels to be rendered by TOPCAT for only two spectra and their overlapping images. In TOPCAT, it is now possible to directly compare measured flux densities with their uncertainties for images and spectra in a 4D cube with axes: Right ascension, Declination, Wavelength, and Time. The data cube also contains the uncertainty for each pixel measurement, also in flux density. It is possible to export other metadata as a part of the cube, for example, the link to the original FITS file or the relevant spectrum for image regions.

HiSS-Cube supports the visualization by a simple export to FITS or VOTable formats that are readable by tools such as TOPCAT. We can visualize the Galaxy2 4D cube, see Fig. 8.1 so that the spectrum is represented by a line in the center of each galaxy. The image regions have five distinct wavelength coordinates since all five filters were used. Note that the galaxies are much brighter in the *g, r, i* filters in the middle and fainter in the *u* and *z* filters at the edges.

It can also be visualized in a different view to see the uncertainties (sigma values) with the same axes. The graphical artifacts in Fig. 8.1 are caused by imperfect anti-aliasing since there are lots of image regions drawn at the same spectral coordinates (in fact, Fig. 8.1 shows a superposition of 185 image regions captured at different times). Note that the image regions in the *u* and *z* bands on the edges have higher uncertainty than the regions in the *g, r, i* bands in the middle. This is because the galaxies are fainter in those wavelengths and the measurements are more uncertain.

It is possible to rotate and slice the cube to compare other dimensions as well, see Fig. 8.2.

In the end, we did not use TOPCAT for visualization of big data because it is not specialized for multi-dimensional data cubes, and the analysis within is therefore not very straightforward. As we show in Fig. 6.3, we used `H5Web` tool instead, which provides higher flexibility. However, the visualization in TOPCAT shows how easy it is to use IVOA standards on top of HiSS-Cube due to convenient compatibility between `h5py` and `astropy` libraries since both use similar Python data structures.

[1] http://www.ivoa.net/documents/

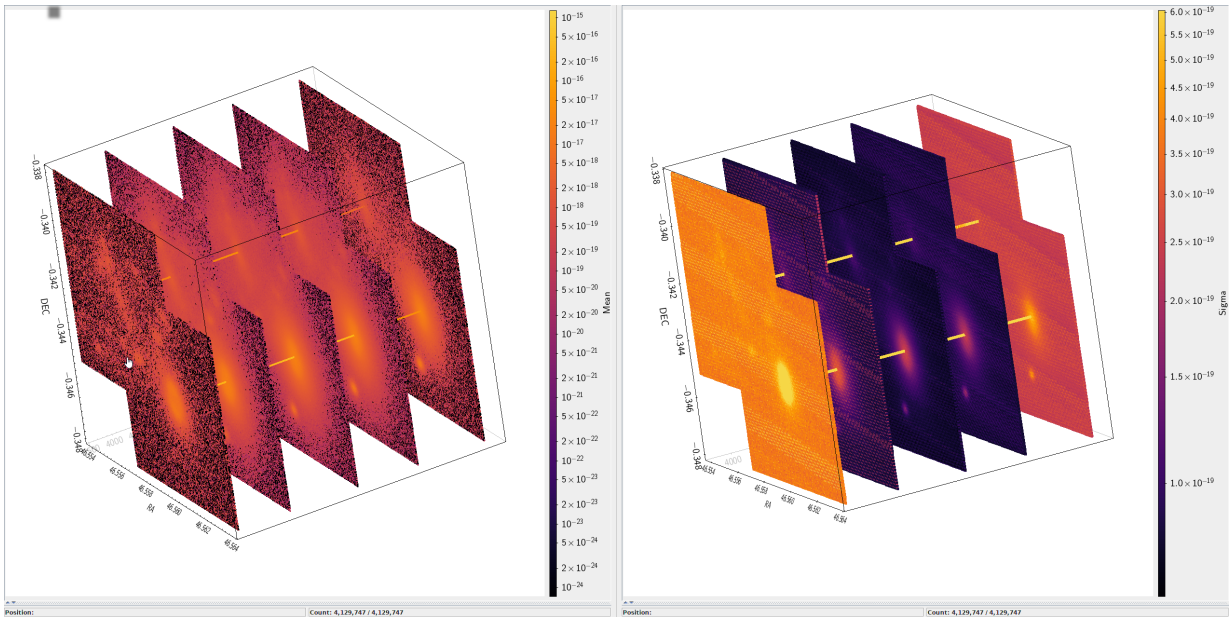

Figure 8.1: Screenshot of Galaxy2 dataset exported to VOTable and visualized in TOPCAT. The view on the left-hand side depicts the mean values and the view on the right-hand side the sigma values. Each pixel is therefore described by a Gaussian distribution with these two parameters.

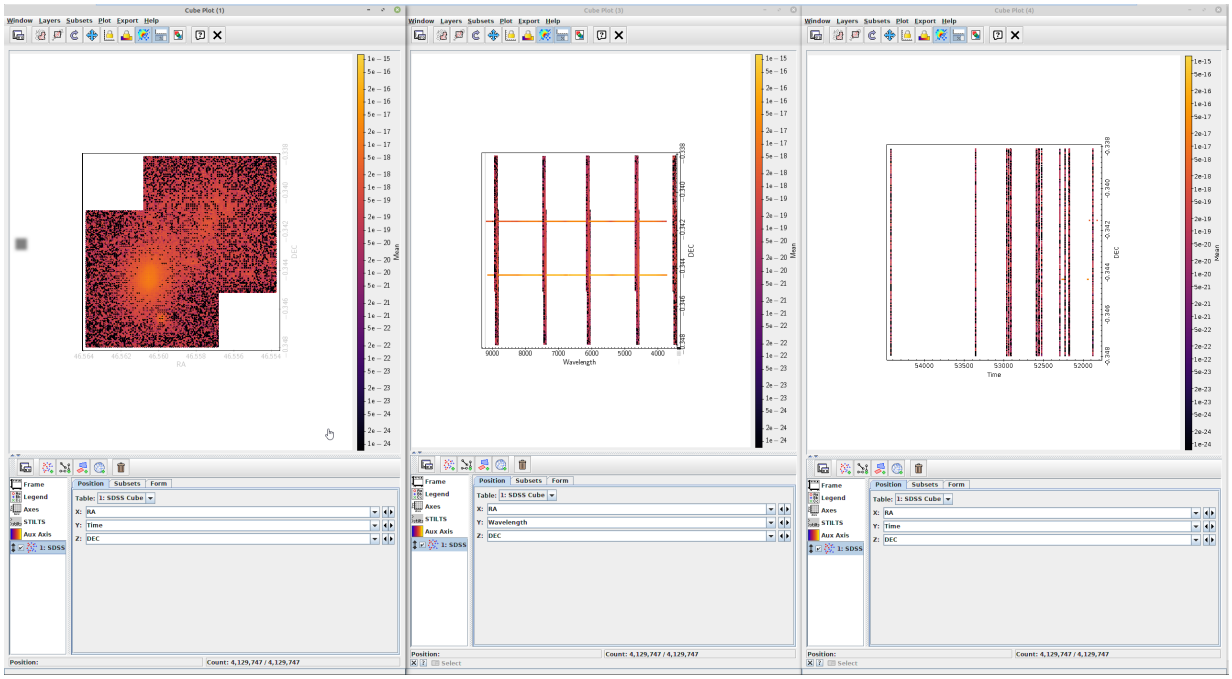

Figure 8.2: Three rotated views of the Galaxy2 dataset. All three views plot the mean values. In the view on the left, the images are seen from the front (Right ascension and Declination axes are visible). In the middle view, the two horizontal lines represent the two spectra at the centers of their relevant image regions and the vertical lines are images at five distinct spectral coordinates (Declination and Wavelength axes are visible). In the view on the right, the spectra observation times are compared to the observation times of image regions. The images are vertical lines and the spectra are represented by dots at the same declination as on the view in the middle (Declination and Time axes are visible). The spectral axis is displayed in Angstroms and the time axis in Modified Julian Date.

8.4 Performance - Database construction

The steps needed for *Database construction* have been described in Section 4.1. We have optimized reasonably the slowest parts of the code like initializing the WCS programmatically or preventing any unnecessary copies of `NumPy` arrays. The code would be probably faster in Cython[2], but since there are in total nine numerical operations that need to be calculated for every pixel of an image just to extract the uncertainties, it would not be faster by orders of magnitude. Moreover, we want to prove that we can maintain HiSS-Cube purely in Python, without suffering from the *two language problem* (Python + C code maintenance). We rely on the community to maintain the optimized C-language parts of the 3rd party libraries.

The average sequential *Database construction* times for both the images and the spectra are listed in Fig. 8.3. Based on these times it can be simply calculated how long it takes to construct the database for the whole Stripe 82 (which is only approximately 20% of the overall SDSS data volume), see Table 8.1. Without HiSS-Cube, the preprocessing would need to be run every time the data is combined. However, the HiSS-Cube enables to use the region references and they can be reused for any data combination needed. Therefore, the preprocessing can run only once within the *Database construction* of the HiSS-Cube and be reused for any database queries, be it for visualization or machine learning purposes.

Table 8.1: Sequential *Database construction* times for the Stripe 82 spectra and images. The 638 total hours are approximately 26 days of sequential processing time.

	Images	Spectra	Total
Number of	987360	129373	N/A
Average unit preprocessing time [s]	2.18	1.11	3.29
Average total preprocessing time [h]	**599**	**40**	**638**

[2] `https://cython.org/`

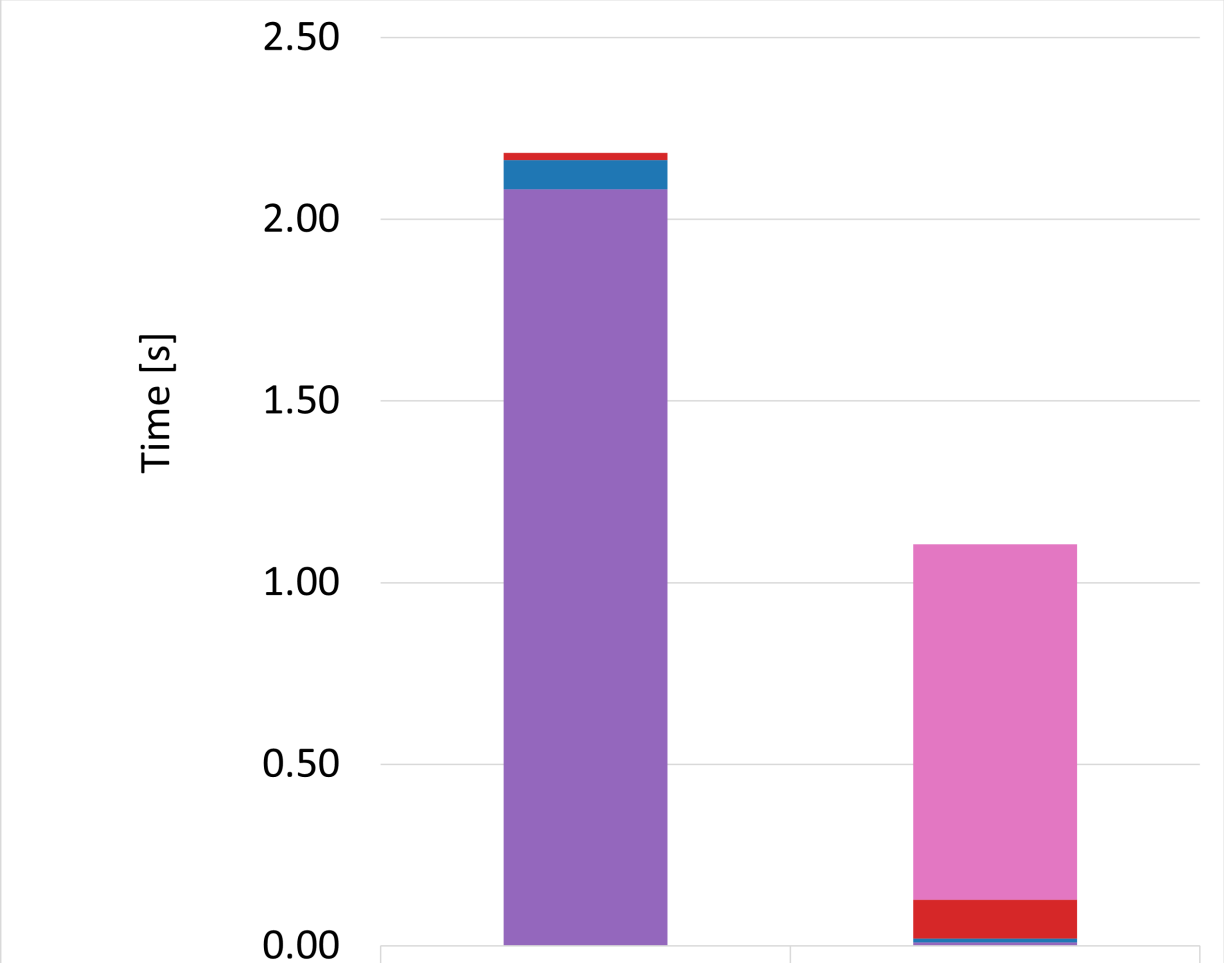

	Image	Spectrum
Linking	0.00	0.98
Metadata	0.02	0.11
Metadata cache	0.08	0.01
Data	2.08	0.01

Figure 8.3: *Database construction* times for one image and one spectrum in seconds. The dominant phase for images is the *Image data* phase, from which the majority are numerical operations. For the spectra, the dominant phase is the *Linking data* phase, from which the vast majority is finding the image regions and computing the image region references.

8.5 Performance - Compression efficiency

Table 8.2 lists compression ratios of HDF5 and FITS format used in the SDSS survey measured on Galaxy2 data. In the SDSS reduction pipeline, the FITS datasets are compressed by Float lossy compression followed by BZIP2 compression. Therefore, to be directly comparable, HiSS-Cube also uses the same Float lossy compression. The HiSS-Cube File data are twice as big as the FITS data after the Float compression because they contain mean and sigma values for every pixel. In our experiments, the BZIP2 was not performing well with the chunked HDF5 datasets. The best results were achieved with BitShuffle followed by GZIP compression. The compression ratio is calculated as Float+BitShuffle+GZIP / Float for the HDF5 and as BZIP2 / Float for the FITS. The reason why we cannot perform a comparison with the original FITS images is that the Float compression is not reversible, therefore, FITS Uncompressed is stated as N/A. The FITS images also do not contain any MIP Maps.

For the Galaxy2 dataset, the Dense Cube Spectral + Image regions stored query results (see Fig. 5.1) represent cca 5% of Semi-Sparse Cube in Float compressed format and cca 2.5% in Float+BitShuffle+GZIP format.

We can conclude that HDF5 achieves similar compression ratios as FITS for full-resolution data (denoted as No MIP map data in Table 8.2) and that HDF5 compression ratio remains almost unchanged if applied to full-resolution data extended with low-resolution MIP map data.

Table 8.2: Compression ratios of HDF5 and FITS with the Galaxy2 dataset.

	FITS				**HDF5**			
	Uncompressed	Float compressed	BZIP2 compressed	**Compression ratio**	Uncompressed	Float compressed	Float BitShuffle GZIP	**Compression ratio**
No MIP map data	N/A	2.2GiB	587 MiB	**27%**	9.1GiB	4.4GiB	1.3GiB	**29%**
With MIP map data	N/A	N/A	N/A	**N/A**	12GiB	5.8GiB	1.8GiB	**30%**

Table 8.3: Performance of visualization query run on the Galaxy2 dataset.

	FITS		HDF5		**Speedup**		
	Float compressed	BZIP2 compressed	Float compressed	Float BitShuffle GZIP	**HDF5 Float / FITS Float**	**HDF5 GZIP / FITS Float**	**HDF5 GZIP / FITS BZIP2**
Number of read ops.	16120	1350578	22803	19173	N/A	N/A	N/A
Amount of data accessed	40 MiB	10 GiB	408 MiB	140 MiB	N/A	N/A	N/A
Time (s)	35.91	1350.8	31.15	25.61	**1.15**	**1.40**	**52.75**

Table 8.4: Performance of Global database query run on the Galaxy2 dataset.

	FITS		HDF5		**Speedup**		
	Float compressed	BZIP2 compressed	Float compressed	Float BitShuffle GZIP	**HDF5 Float / FITS Float**	**HDF5 GZIP / FITS Float**	**HDF5 GZIP / FITS BZIP2**
Number of read ops	16120	1350578	6	13	N/A	N/A	N/A
Amount of data accessed	40 MiB	10 GiB	220 MiB	33 MiB	N/A	N/A	N/A
Time (s)	35.91	1350.8	0.32	2.38	**113.93**	**15.14**	**569.51**

8.6 Performance - speed and I/O efficiency

This section reports on measurements of processing time and I/O performance. By I/O performance we understand the amount of read operations and the total amount of data accessed on disk. The measurements were made on the Galaxy2 datasets in both HDF5 and FITS formats. For verification purposes, we wrote all output data into a FITS file that can be visualized in TOPCAT and a sample of it is accessible on Zenodo, see Chapter B. We do not consider the time needed to write the exported data into the FITS file because it is outside of the HiSS-Cube framework.

8.6.1 FITS baseline

As a baseline for speed and I/O efficiency testing, we implemented a trivial algorithm (hereinafter **FITS baseline**). It iterates through FITS files in two folders - `spectra` and `images`. This program reads the header of the spectrum and then goes through all files in

the `image` folder, reads their headers, and checks whether the image overlaps the spectrum. If so, it cuts the image out and submits it to TOPCAT in the same way as the HiSS-Cube does. However, to even the odds up, we have introduced almost ideal conditions for the FITS baseline - the images form a very dense region around the spectra, meaning there are no FITS files in the `images` folder that do not overlap the spectra. In this regard, the FITS baseline will be close to the performance of a database that knows which images actually do overlap the spectra, because it will not be opening image headers only to find out that there is no overlap. The FITS baseline uses `astropy.io.fits` module with memory mapping enabled.

8.6.2 Performance - Visualization

This section describes the performance of the visualization of the Galaxy2 dataset. We compared the FITS baseline with the Sparse Cube VOTable export, see Fig. 4.1, hereafter referred to as **HDF5 visualization**. The HDF5 visualization displays all spectra with their respective image regions from Galaxy2, see Fig. 8.1 and Fig. 8.2. For visualization performance results, see Table 8.3.

The results of the FITS baseline are listed in the FITS Float compressed and BZIP2 compressed columns. It runs on the FITS Float and BZIP2 compressed data without MIP map data, see Table 8.2. The FITS baseline performs very well on the float-compressed datasets since we introduced the ideal conditions for it. The `astropy.io` memory mapping feature enables it to read directly image region data without scanning the rest of the image. The 40 MiB is the size of spectra and their relevant image regions combined. On the other hand, the FITS baseline performs poorly on the original BZIP2 compressed FITS files of the SDSS survey because there is no possibility of memory mapping them. It also does not cache the already visited images and needs to decompress them again if accessed multiple times. The BZIP2 is well known for being much slower at extracting than GZIP. Since the decompression time dominates, the compression method of the whole FITS file does not matter.

The results of HDF5 visualization are listed in HDF5 Float compressed and Float+BitShuffle+GZIP columns of Table 8.3. It runs on the HDF5 Float and Float+BitShuffle+GZIP data with MIP map data, see Table 8.2. The compression of image datasets in HiSS-Cube File is done on individual chunks and together with the region referencing it enables to read only the compressed data of relevant chunks. Compared to for example FITS tile-based compression [68] we had the synergy with HDF5 region referencing that enables us to only read relevant chunks directly. Moreover, since the HDF5 chunks have a similar size to the image regions, no unnecessary data is read.

The comparison of FITS baseline and HDF5 visualization is listed in the speedup columns of Table 8.3. The HDF5 visualization on uncompressed data is 15% faster and even 40% faster on compressed data, compared to the memory-mapped FITS baseline working in ideal conditions on uncompressed data. This verifies our expectation within the design, see Chapter 4, since the HDF5 indeed needs to read only relevant data before decompressing them (the FITS baseline is not running any decompression). Furthermore,

the comparison to Float compressed performance shows that the HiSS-Cube visualization is more efficient in I/O operations than the memory-mapped `astropy` reader used in FITS baseline while saving cca 70% of storage resources. Since HDF5 exploits much better the spatial locality and efficiently caches the data, it is faster even if the number of read operations and the amount of data read is greater. The reason why HiSS-Cube has more read operations than FITS is due to the combination of spectra and images inside the HDF5 file format. Consequently, the first operation of de-referencing an image region from a spectrum region reference generates 5000-7000 read operations in order to cache the HDF5 file structure.

8.6.3 Performance - Global database query & export

This section describes the performance of the *Global database query* combining all spectra and their relevant image cutouts on the Galaxy2 dataset, see the architecture in Fig. 4.1. Hereafter, we refer to it as *Global database query*. The results of the performance measurements are listed in Table 8.4.

We use the FITS baseline, the same as for visualization, see Section 8.6.1. The FITS baseline is relevant for the *Global database query* as well because both could be used by machine learning algorithms. Machine learning usually processes the whole data and does not need efficient access to small slices of it, in contrast to visualization. The results are listed again in FITS Float compressed and BZIP2 compressed columns.

The results of the *Global database query* are listed in Table 8.4 in HDF5 Float compressed and Float+BitShuffle+GZIP columns. The fastest performance is achieved on Float compressed data since it invokes only one sequential read from the disk along with some metadata operations, in total 6 read operations accessing 220MiB of data. More storage-efficient, but slower, is the run on Float+BitShuffle+GZIP HDF5 file where it needs to read some additional metadata, in total 13 read operations accessing 33 MiB of data.

When comparing the HDF5 Float+BitShuffle+GZIP to FITS Float compressed runs, the HDF5 is 15 times faster than the FITS. Please keep in mind that this is where HDF5 needs to decompress the whole data while FITS does no decompression. The speedup is even higher when the *Dense_3D_cube* is not compressed by lossless compression, only by Float compression.

When using the Float+BitShuffle+GZIP compression on all of the HDF5 datasets except for the *Dense_3D_cube*, the *Dense_3D_cube* requires cca 15% of storage space instead of 2.5%, but then the HDF5 is 114 times faster than the FITS Float compressed baseline.

The speedup is enormous when comparing HiSS-Cube to the FITS baseline on BZIP2 compressed data, the reasons are the same as for visualization.

8.7 Performance - Metadata operations

In this section, we discuss the performance of the *Metadata* phase in the sequential version of HiSS-Cube. Note that the *Metadata cache* phase is not part of the sequential version of HiSS-Cube File structure, these caches are added only for parallelism purposes. We also discuss the reasons for their creation in this section. In summary, the *Metadata* phase reads the images and spectra metadata of the FITS files and writes them into the HiSS-Cube File, including the preprocessing of the metadata that is needed. The most interesting part of this phase is the database index creation which is also the biggest bottleneck for the sequential version of HiSS-Cube caused by its tree-based index structure.

We focus on performance on image metadata as the preprocessing is more complex and better illustrates the bottleneck of the Python implementation.

Let us now describe the biggest bottleneck caused by sequential HiSS-Cube File structure for big data. We compare two approaches:

1. **h5py** - This implementation uses the `h5py` API for the processing of metadata, writing of attributes, and creation of the hierarchical database index.

2. **Python C extension** - This implementation calls the `HDF5 C API` directly. This variant is the same as the `h5py` implementation, but all HDF5 collective operations - writing or creation of groups, datasets, and attributes, are called directly via `Python C extension` from the `HDF5 C API`. We chose the `Python C extension` over `Cython` for its simplicity.

The total running times are depicted in Figures 8.4 and 8.5. These show that even for 300 thousand images the total time needed by approach (1) exponentially increases, reaching over 30 thousand seconds, while approach (2) takes 5 thousand seconds for one million images (the whole dataset).

The reason is shown by comparing Figures 8.6 and 8.7. Fig. 8.6 depicts that it is taking progressively more time to import the metadata as the number of already imported images in the HiSS-Cube File grows. After extensive analysis with HDF5 community, we concluded this is due to design inefficiencies within the `h5py` library in the datasets and group creation operations. When the amount of groups in any HDF5 file grows over tens of millions, a significant overhead arises when trying to create another group or dataset within the `h5py` library. This finding will not result in a fix of the `h5py` though, as the HDF5 itself was not optimized for such a huge amount of entries, but rather having a few huge datasets.

We can confirm this issue is not present in the `HDF5 C` library in Fig. 8.7. There it is shown that even though there are random slowdowns occurring when the amount of objects within HiSS-Cube File grows, asymptotically the rate remains constant. That means the total running time shown in Fig. 8.5 scales linearly with the number of images imported. Therefore, by using the C API directly we have circumvented the problem.

After this significant speedup, we experimented with parallel implementation using the C API directly. However, there the issue of HDF5 optimization came back. Since it was

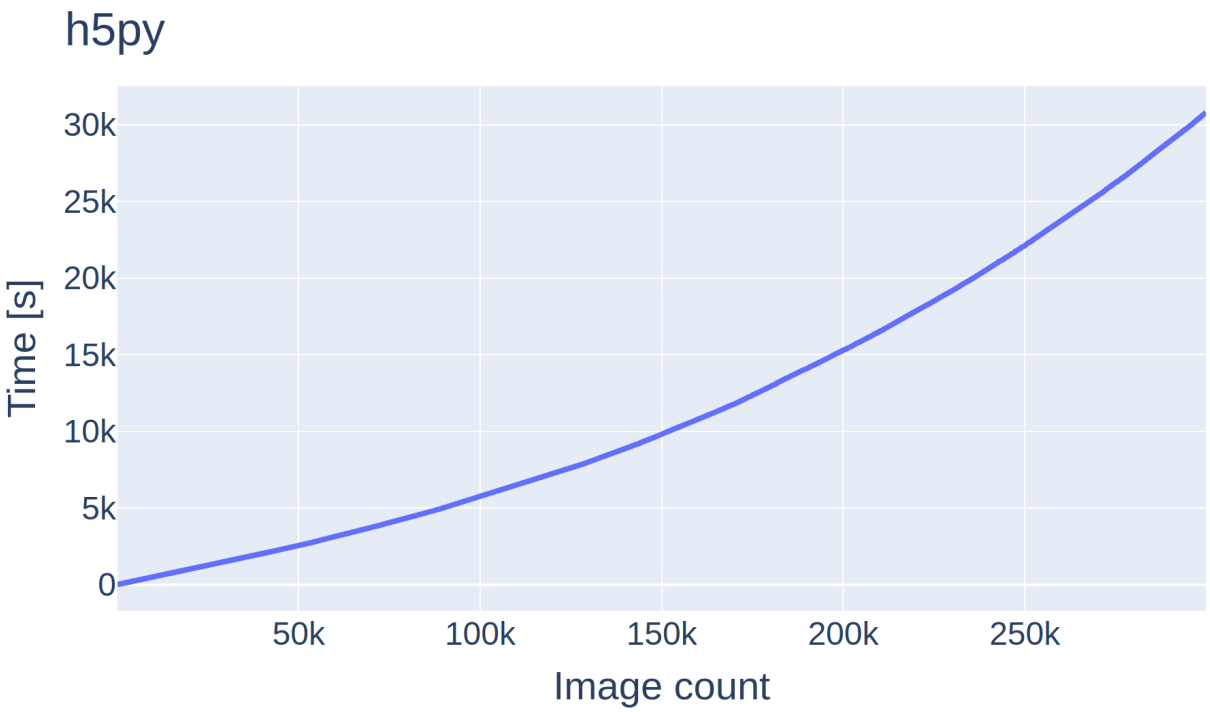

Figure 8.4: The total running times of HiSS-Cube *Metadata* phase running on the `h5py` Python interface. The x-axis represents the amount of images already imported and the y-axis total time in seconds.

rather designed for a small number of entries which have potentially unlimited size, we found no way how to use it with such a high amount of entries. For the full test data, there were tens of millions of datasets (number of images or spectra times the down-sampled resolutions) and hundreds of millions of groups in the database index, depending mostly on the spatial index fidelity.

For this reason, we forsake the idea of having the HiSS-Cube File human-readable within any HDF5 reader in favor of performance and redesigned its structure to its flat parallel version depicted in Fig. 6.1. We describe its performance in the next chapter.

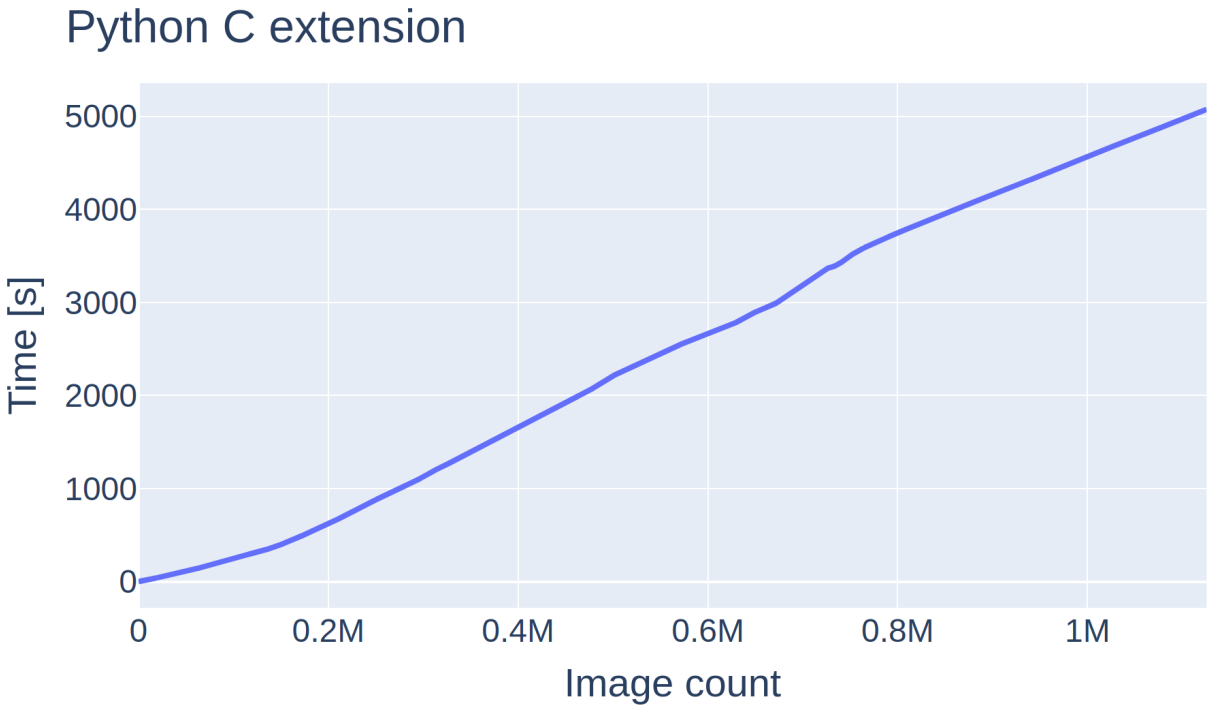

Figure 8.5: The total running times of HiSS-Cube *Metadata* phase running on the `Python C extension` interface using `HDF5 C API` directly. The x-axis represents the amount of images already imported and the y axis total time in seconds.

Figure 8.6: The speed of HiSS-Cube *Metadata* phase running on the `h5py` Python interface. The x-axis represents the amount of images already imported and the y-axis time needed to import metadata of 100 images in seconds.

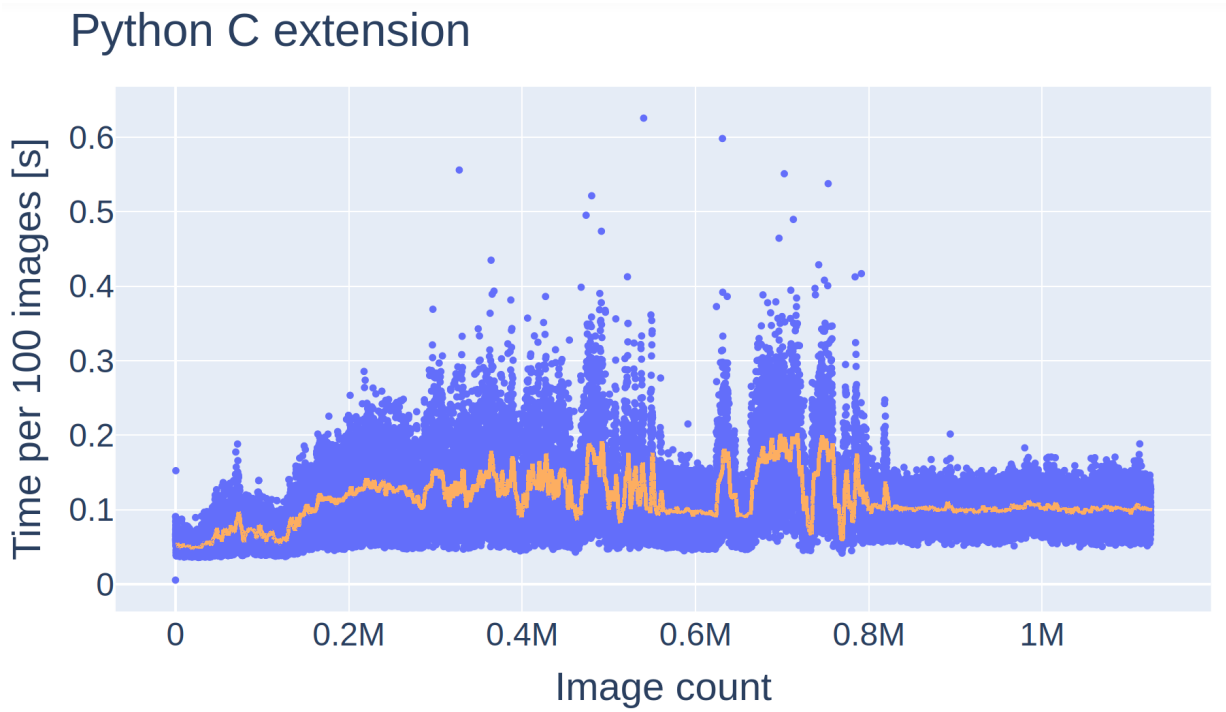

Figure 8.7: The speed of HiSS-Cube *Metadata* phase running on the `Python C extension` interface using `HDF5 C API` directly. The x-axis represents the amount of images already imported and the y-axis time needed to import metadata of 100 images in seconds.

Chapter 9

HiSS-Cube parallel performance evaluation & scalability

In this chapter, we primarily discuss the parallel performance of HiSS-Cube, further focusing on parallel write performance.

9.1 HPC Platform & Hardware

All experiments were performed on the Karolina HPC[1]. Karolina HPC has 24 physical I/O nodes (SSU-F – ClusterStor controllers), each of which is used by one Lustre Object Storage Server (OSS), which further uses two Object Storage Targets (OSTs) per OSS, totaling 48 Lustre OSTs that serve as independent parallel I/O channels.

The Karolina HPC cluster has 720 compute nodes with 2x AMD 7H12, 64 cores, 2,6 GHz (128 CPU cores per node, 92,160 CPU cores in total), 256 GB RAM per node, and a 100 Gbit Infiniband interconnect between the nodes and the file system. All nodes have Infiniband connections to the file system.

Storage is the bottleneck in our case, and we will describe it in more detail. It has a total capacity of 1361 TB on the `Lustre` file system, with a theoretical write bandwidth 730 GB/s and a read bandwidth 1198 GB/s. These values cannot be reached when the cluster is under load, and we used our own benchmark described in Sec. 9.1.1. The majority of the storage capacity is delivered by 24x SSU-F - ClusterStor 2U24 Scalable Storage Unit Flash Storage Controllers, consisting of 24x Cray ClusterStor 3.2TB NVMe x4 Lanes Mixed Use SFF (2.5in) U.2 with Carrier and 4x Cray ClusterStor InfiniBand HDR/Ethernet 200Gb 1-port QSFP PCIe4 Adapter (Mellanox ConnectX-6).

[1] `https://www.it4i.cz/en/infrastructure/karolina`

9.1.1 Karolina I/O performance benchmark

The IOR[2] tool was used to test the performance of the Karolina file system. We chose the file-per-process benchmark because it provides the best performance. We averaged the performance over multiple runs. While the compute nodes are dedicated to the job, the file system is shared among all jobs in the cluster.

The measured average file-per-process write performance was approximately 7 GB/s per compute node. This is reasonable, given that the theoretical maximum bandwidth of the Infiniband connection is 12.5 GB/s. We use the average write performance for better reproducibility of the results because an average user will not have dedicated HPC hardware available, at least not in terms of storage resources. The Infiniband connection is the limiting factor for I/O bandwidth up to approximately 96 compute nodes, where the overall parallel bandwidth limit of the `Lustre` file system begins to dominate.

9.1.1.1 Karolina `Lustre` limitation

We must consider an additional aspect that influenced our measurements. The software configuration of the Karolina HPC `Lustre` file system degrades the performance of shared writes (even independent ones) to the same file. We assume that this setting is there for fairness reasons. The theoretical bandwidth numbers above are unachievable because that would severely compromise the fairness of resource allocation on the cluster for other jobs. The detailed behavior of scaling on one compute node is presented in Table 9.1.

To summarize it, shared writes do not scale with the **stripe count**[3] on one compute node, that is, with the number of `Lustre` OSTs that the compute node has available as parallel I/O channels. Only adding more compute nodes to the job unlocks a higher bandwidth available for independent writes to a shared file. Therefore, to reach high I/O bandwidth, the job must allocate a high number of compute nodes.

According to our testing, this software configuration does not limit the I/O operations per second (**IOPS**); therefore, random read and write operations are not limited on one compute node. Therefore, the IOPS-heavy *Parallel database construction* phases (which do not need high bandwidth) are able to saturate the performance of the underlying parallel filesystem with fewer compute nodes.

[2] `https://ior.readthedocs.io/en/latest/`

[3] `https://wiki.lustre.org/Configuring_Lustre_File_Striping`

Table 9.1: Write Bandwidth and Bandwidth/Compute Node in [MB/s] depending on the method used (File-per-process vs. Shared file) and the stripe count used in the `Lustre` file system beneath. The bandwidth does not scale with additional writers or stripes on one compute node, but it still does scale with one stripe and an increased number of compute nodes.

Method	CPUs/node	Nodes	stripe_cnt	Bandwidth	Bandwidth/node
File-per-process	32	8	32	67214	8402
Shared file	32	1	32	868	868
Shared file	1	1	1	921	921
Shared file	1	1	2	916	916
Shared file	2	1	1	1236	1236
Shared file	2	1	2	1240	1240
Shared file	4	1	1	1180	1180
Shared file	4	1	2	1178	1178
Shared file	8	1	1	1015	1015
Shared file	8	1	2	1020	1020
Shared file	1	2	1	1562	781
Shared file	1	4	1	3827	957
Shared file	1	8	1	6238	780
Shared file	1	2	2	1722	861
Shared file	1	4	2	3471	868
Shared file	1	8	2	6987	873

9.2 Testing data

We used the SDSS survey for both the images and spectra. For the images, we used Data release 8 [2], which has approximately 4.5 million images, resulting in approximately 60 TB of uncompressed data. This data does not contain precomputed errors; therefore, it grows to approximately 120 TB after data enrichment is performed by the *Image data* phase. For the spectra, we used Data release 14 [1], which has approximately 5 mil. of spectra, resulting in approximately 960GB. The number of images is already comparable with the big data we want the HiSS-Cube to scale to, such as the LSST [44] or the Nancy Grace Roman Space Telescope [41], but the sizes of the images will be much larger for these surveys. Therefore, we focused on the overall *Parallel database construction* bandwidth in our performance tests. For the detailed processing times see Sec. 9.4.

The reasons why we chose this data for parallel scalability testing are the following:

1. The providers of data from PB-scale astronomical surveys usually do not provide sufficient bandwidth to transfer the entire survey over the Internet in a reasonable time.

2. Currently, we do not have access to an HPC system with PB-scale storage capacity. Therefore, we used fast HPC storage with a smaller capacity and extrapolated its scalability to the PB-scale. Using fast HPC storage also allowed us to conduct many

more experiments in the same amount of time, compared to slower storage with higher capacity.

9.3 Performance - Parallel HiSS-Cube database construction & querying

In this chapter, we primarily discuss the write performance of HiSS-Cube.

Our key requirement is high sequential read performance; for example, if the entire *Dense_3D_cube* dataset needs to be read repeatedly. However, this is not an issue because the sequential read performance is already well proven for HDF5 [22] [31], which is also why it was chosen for the HiSS-Cube implementation in the first place. The random read performance is also important for *Parallel database construction* and *Global database query*, but it is very good, as explained in this chapter. The most limiting factor is the sequential write performance of HiSS-Cube or, rather, of the underlying parallel filesystem; therefore, we focus mainly on it in this chapter.

All phases of *Parallel database construction* are good examples of I/O-bound computations. This is because all CPU computations within the phase (down-sampling, error reconstruction, metadata recalculation, etc.) have a linear complexity with the volume of input data. Because all the phases are fully parallelized within the Master/Worker model, they have linear speedup with an increasing number of CPU cores. There is one exception to this rule: the *Linking data* phase, which for each spectrum finds overlapping images in $O(Nlog(N)))$ time, where N is the number of images. However, this phase needs to access the *Image_DB_index* to do so within the HiSS-Cube File, and therefore, is again I/O-bound. The *Parallel database query* is I/O-bound for similar reasons.

Therefore, **HiSS-Cube will always be I/O-bound** by the Infiniband connection limit of one compute node, or by the underlying parallel file system limits.

The relative sequential *Database construction* time complexities per image and spectrum were already shown in Fig. 8.3. SDSS data (described in detail in Sec. 9.2) contain approximately 4.5 mil. images and 5 mil. spectra. If multiplied by the computing times from Fig. 8.3, it gives approximately 177 days of sequential processing time for the *Database construction*.

Because we succeeded with the full parallelization of the *Parallel database construction*, the HiSS-Cube performance is now limited by I/O performance rather than computational performance. Specifically, it is limited by the write performance of the underlying file system, as shown in Sec. 9.1.1.

9.4 Parallel results

Fig. 9.1 shows that all the phases scale linearly as the number of CPU cores increases. Let us reemphasize that all phases are I/O-bound. Therefore, this chart shows how the saturation of the parallel file system evolves with an increasing number of CPU cores or

compute nodes, and not how the computational times scale. For example, the fact that the *Image data* phase takes the majority of the time is not caused by the phase spending more time in computations but rather by waiting longer for the I/O. We can also observe that the *Global database query* accounts for a significant portion of the overall running time. This is a good result, as we need to keep in mind that this query needs to go through all the data within the database, for 5 million spectra it finds overlapping image regions in up to 4.5 million images.

We further focused on *Parallel database construction*. In Fig. 9.2, we can see that the *Image data* phase takes the most time. The *Image data* phase does not perform a high number of I/O operations per second; rather, it reads and writes large amounts of data. Its CPU speedup is linear, and the I/O speedup for this phase is also linear before it reaches the maximum bandwidth of the file system, which occurs if the number of compute nodes increases over approximately 72 (9216 CPU cores, see the last bar in Fig. 9.2). More specifically, the performance of the *Image data* phase in images processed per second is bounded by the read and write bandwidth performance of the underlying file system and not by the I/O operations per second.

The *Linking data* phase constitutes another significant portion of the overall processing time. However, its time complexity is not dependent on the data volume, but rather on the number of images overlapping the spectra and the number of spectra (see Sec. 6.3). While spectroscopic surveys can span hundreds of millions of spectra (GAIA data release 3 [11]), which is two orders of magnitude higher than our test data, imaging surveys will not have more than approximately 1000 observations of the same region of the sky (e.g., LSST will have approximately 1000 observations over a period of 10 years). Our test data contained approximately 200 image observations in the densest areas of the survey. Therefore, the *Linking data* phase is not a bottleneck for scaling toward PBs of such data.

The other phases do not speed up that well with an increase in the number of CPU cores or compute nodes. This is because they are much more I/O-bound in terms of I/O operations, per second and the underlying file system performs well, even for a lower number of compute nodes. The total running time for these phases was in units to tens of seconds. These phases do not read or write large amounts of data; therefore, they are not limited by the Infiniband connection of the compute node but rather by the performance of the `Lustre` file system. Therefore, we can saturate the I/O limits with a lower number of compute nodes. Moreover, we do not need the performance of these phases to scale better because their complexity is scaling with the number of images and spectra, rather than their data volume. As shown in the paragraph above, these numbers will not increase significantly for the big data we want to scale to.

However, we wanted to scale the *Image data* phase towards PB-scale surveys. These will not have orders of magnitude higher number of images, but instead of 16 MB images, such as our test data, there can be fully calibrated images with sizes of tens of gigabytes. Let us now focus particularly on the scalability of the *Image data* phase as it is a bottleneck and could be an even larger one in the future.

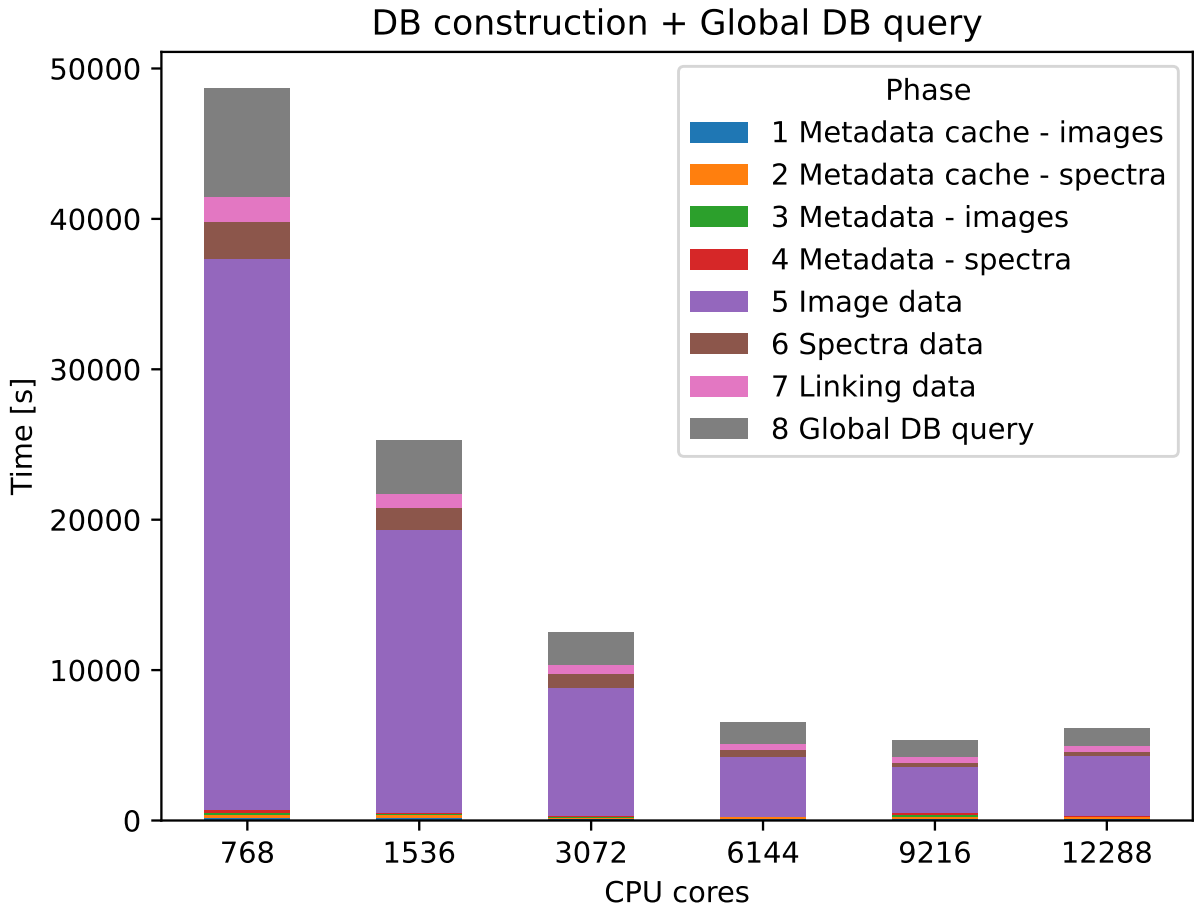

Figure 9.1: The HiSS-Cube running times of the individual phases from Fig. 6.2 (including the *Global database query*) for increasing numbers of CPU cores.

9.4.1 I/O concentrator performance

HDF5 version `1.14` has a new Virtual File Driver called `Subfiling` VFD. This file driver splits the physical `.h5` file into equally sized subfiles that the driver treats as the same logical file, but physically, these are separate files on the file system. All I/O into these files occurs through I/O concentrators, which aggregate all I/O requests from other processes. Typically, there is one I/O concentrator per compute node. A similar functionality has already been implemented in MPI I/O, but having it on the HDF5 layer helps with tuning for a particular application, such as HiSS-Cube. I/O concentrators usually pay off with more than approximately ten thousand CPU cores participating in I/O.

As shown in Fig. 9.4, the write performance of the *Image data* phase with I/O concentrators scales better. However, it comes at a high cost of performance degradation for *Parallel database query* times, as seen when comparing Fig. 9.3 and Fig. 9.1. This is because of the poor read performance of the `Subfiling` VFD in the current version of the HDF5. However, we believe that this issue can be resolved, and we are already cooperating

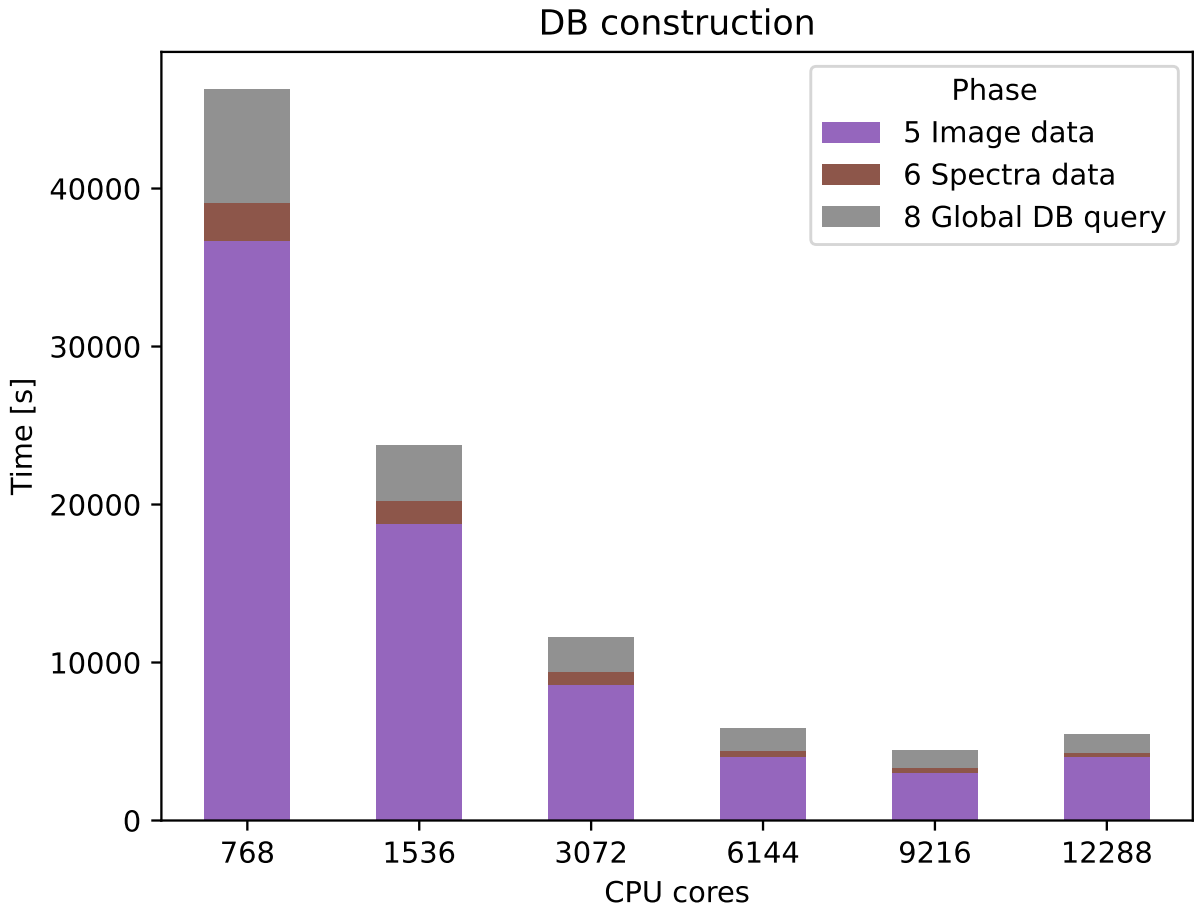

Figure 9.2: The HiSS-Cube running times without *Global database query* phase. This figure shows the same graph as Fig. 9.1, but without the *Global database query* phase.

with the HDF5 community to resolve it.

The detailed *Parallel database construction* is illustrated in Fig. 9.4. Using the `Subfiling` VFD further improves the scaling and shows that the performance of the *Image data* phase is still improving on 144 compute nodes (which is 20 % of the whole cluster - 18432 CPU cores). This is because using I/O concentrators avoids the high I/O contention that occurs when every worker tries to write independently, as shown in Fig. 9.2. The other phases do not scale well with this high number of CPU cores; however, this is again caused by the worse read performance of the `Subfiling` VFD, which can be solved.

9.4.2 Image data performance

In Fig. 9.5, we show the detailed scaling of *Image data* phase when the I/O concentrators are used. The key observation is that the ratio of HiSS-Cube performance compared to the maximum `Lustre` performance is increasing and does not hit the limit of the file system

up to 144 compute nodes, whereas the File-per-process performance does not scale over approximately 96 compute nodes.

9.4.3 Further scalability

Our results have been measured on up to 144 compute nodes, which is 12288 compute cores and 20% of the overall Karolina capacity. We tried to run the tests for bigger portions of the cluster, however, we were not able to finish them for 30% capacity anymore. The results were more chaotic, having two orders of magnitude differences in the overall bandwidth achieved by the system. The bigger the jobs are, the more they are influenced by other load on the cluster since they try to consume bigger portions of the available I/O bandwidth.

As seen in Fig. 9.5, the maximum bandwidth can be achieved with only approximately 13% of the cluster allocated to the job, meaning the cluster is actually optimized to spend approximately 87% of the time in the computation and only 13% in the I/O operations. Therefore, using 30% of the cluster capacity for saturating I/O has rather unpredictable results.

The reason is that the Karolina HPC (and to our knowledge, bigger HPCs as well) are not optimized to handle jobs that consume more than 30% capacity of the cluster just for I/O operations, especially shared write operations to a single file that are even more demanding in terms of shared resources. In addition to high queuing times, the MPI initiation of the processes often crashed even before the HiSS-Cube application entered its own logic, and precious computing time was wasted. It was even more accented by the job often crashing the nodes and rendering big portions of the HPC cluster offline.

Since this was also putting extensive load on the IT4I support we abandoned the testing for bigger portions of the cluster. However, the scalability is proven well by testing on up to 20% capacity of the cluster. The HiSS-Cube will still scale for bigger HPC clusters of similar design to Karolina because the job will occupy a smaller portion of the cluster while having more I/O resources available. We also discuss the further scalability on bigger HPCs in Chapter 10

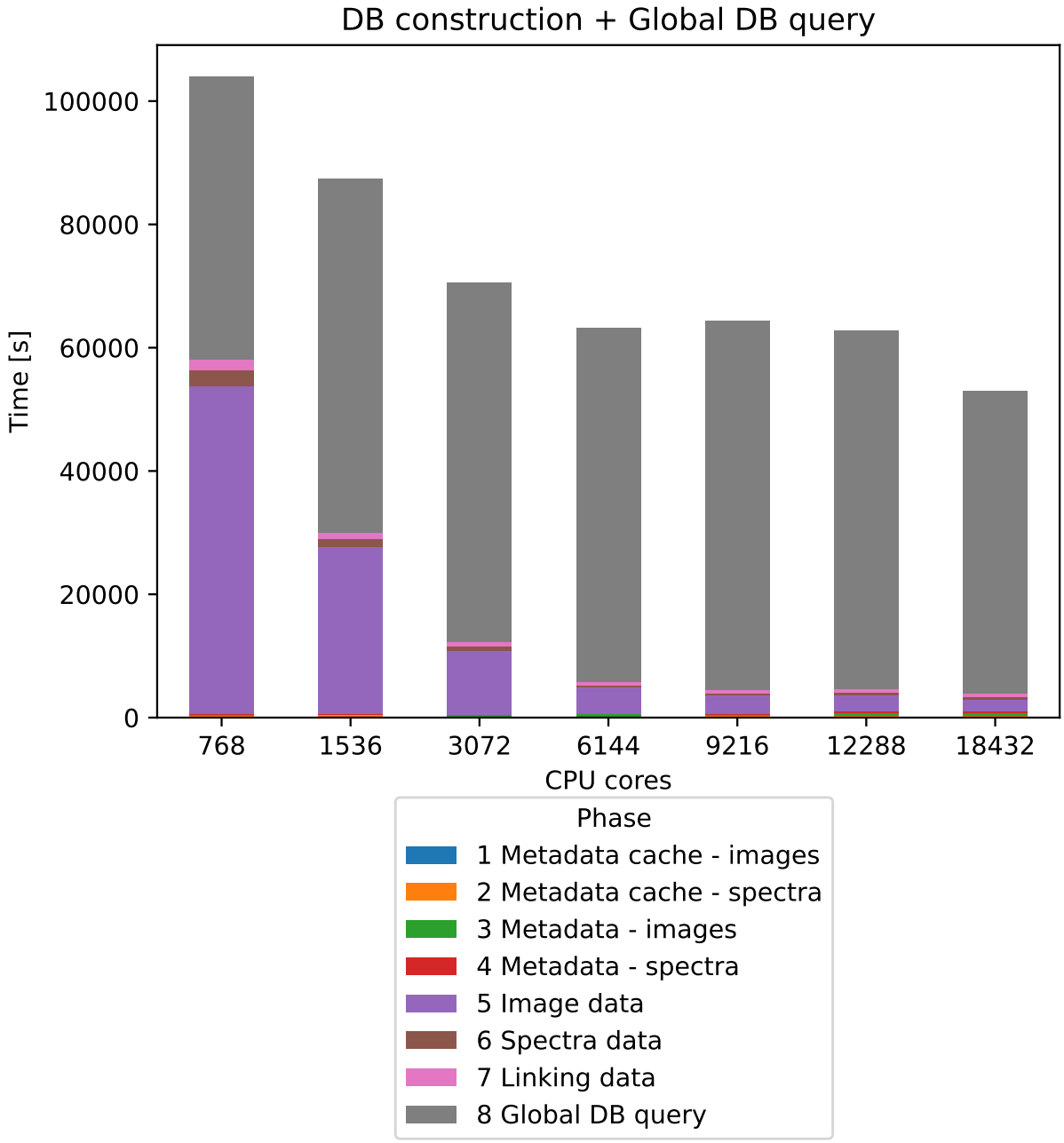

Figure 9.3: The HiSS-Cube running times with I/O concentrators. This figure shows the same graph as Fig. 9.1, while using the I/O concentrators instead of each worker writing independently. It uses 1 I/O concentrator per node with a subfile stripe size of 4MB and the number of subfiles equal to the amount of I/O concentrators.

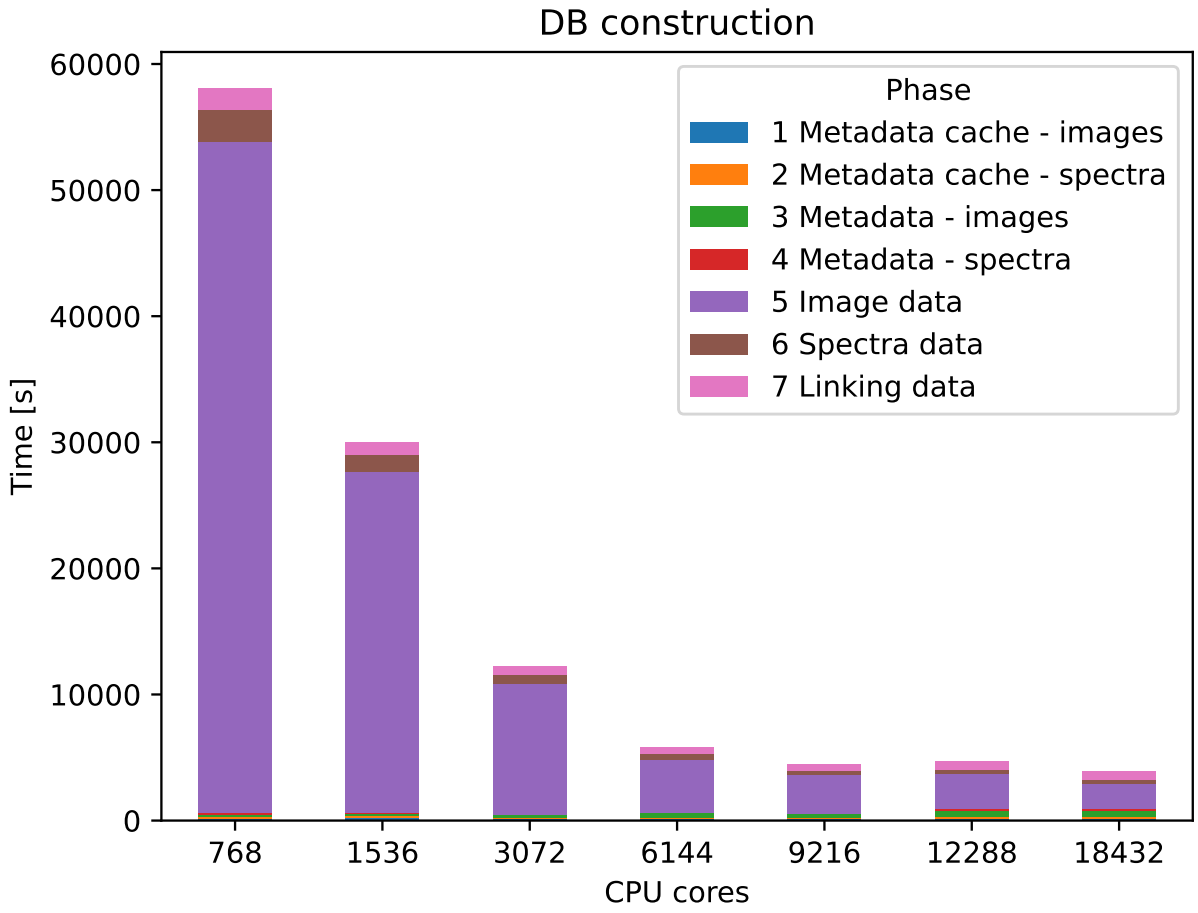

Figure 9.4: The HiSS-Cube running times without *Global database query* phase with I/O concentrators.This figure shows the same graph as Fig. 9.3, without the *Global database query* phase.

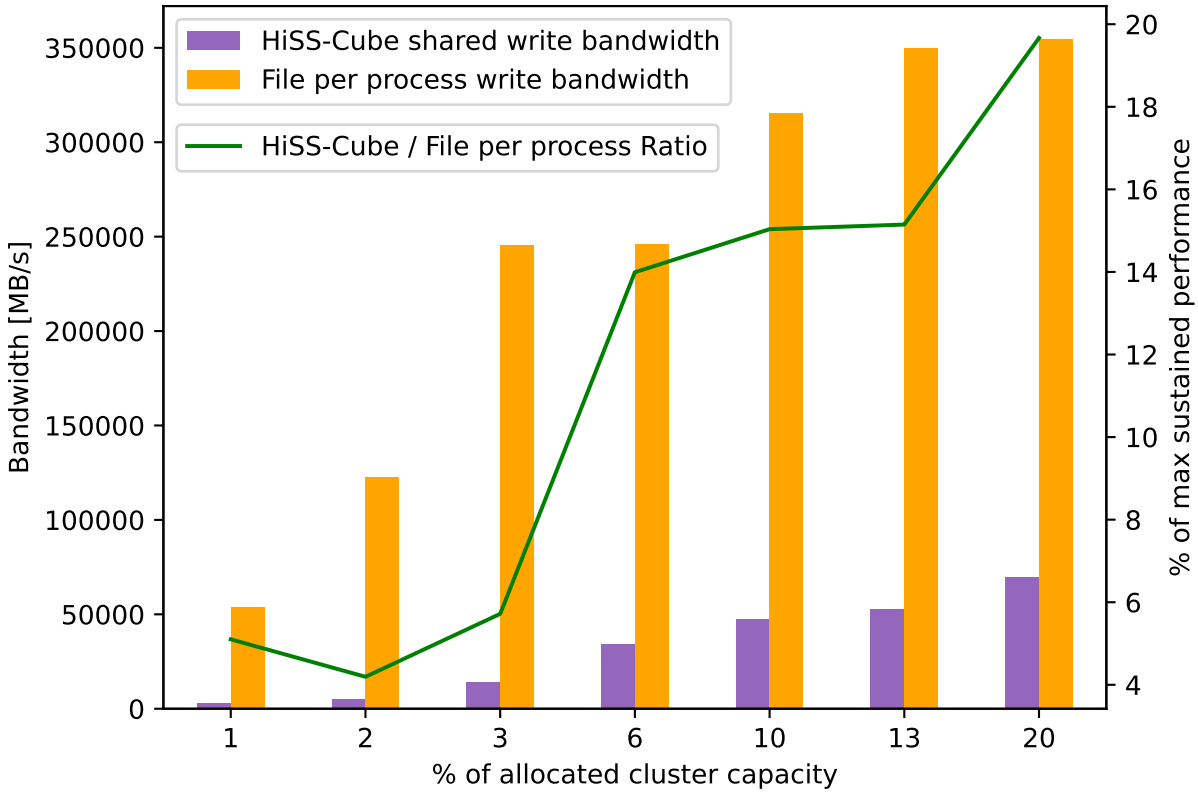

Figure 9.5: The performance of HiSS-Cube *Image data* phase compared to the maximum performance of the underlying filesystem. The bars belong to the left y-axis (Bandwidth in MB/s) and the line chart belongs to the right axis and displays the ratio between the HiSS-Cube write bandwidth and the maximum File per process write bandwidth. On the x-axis is the ratio of allocated compute nodes compared to the total number of compute nodes.

9.5 I/O performance

This section describes the overhead and how it changes with the amount of CPU cores for different phases of the HiSS-Cube processing. No data is being transferred between processes in any of the phases. The only content of the MPI messages is the paths to files or paths to HDF5 datasets. Also, since the MPI communication is used only for workload distribution from the master process to workers and the batch size is configurable, the total number of MPI messages is kept low. Therefore, when the parallel efficiency of HiSS-Cube degrades with a higher number of CPU cores or compute nodes, it is not due to inter-process communication overhead. For profiling I/O waiting times, we have used the `darshan`[4] I/O characterization tool.

With a higher amount of compute nodes, there are more I/O resources available in terms of shared read/write operations on a single file. However, the maximum File per process write bandwidth of the underlying filesystem is already reached with approximately 13% (see Fig. 9.5) of the cluster allocated capacity, recall the specification and limitations of the Karolina supercomputer in Sec. 9.1.1.

Let us now describe the HiSS-Cube I/O performance in the three following phases that take a significant amount of the overall processing time (see Fig. 9.6):

1. *Image data* phase
2. *Spectra data* phase
3. *Parallel database query phase*

The detailed results are shown in Fig. 9.7, Fig. 9.8, and Fig. 9.9. We measure the cumulative time spent waiting for I/O read operation (*Read time*), cumulative time spent waiting for I/O write operation (*Write time*). The rest of cumulative time is described as *Other* in the legend. *Other* includes the time spent in numerical computations and MPI communication, plus it also includes the waiting time for all workers to finish their batch at the end of the communication.

This is best illustrated on the MPIO HDF5 driver because it already stops scaling for some of the phases between 9 and 12 thousand CPU cores and we explain why it happens. The other phases do not spend any significant time in I/O operations and their performance is already satisfactory.

Fig. 9.7 shows that the *Image data* phase with the MPIO driver hits the limit of the underlying filesystem performance with 6144 CPU cores (48 compute nodes). For 9216 CPU cores (72 compute nodes) its Write performance is worse compared to 12288 CPU cores (96 compute nodes), because the compute nodes do not use efficiently the number of 48 available `Lustre OSTs` which are the independent parallel I/O channels (72 is not dividable by 48). The time spent in read operations is negligible. The conclusion together with the observations from Fig. 9.5 is that the limiting factor is the I/O bandwidth available

[4] `https://www.mcs.anl.gov/research/projects/darshan/`

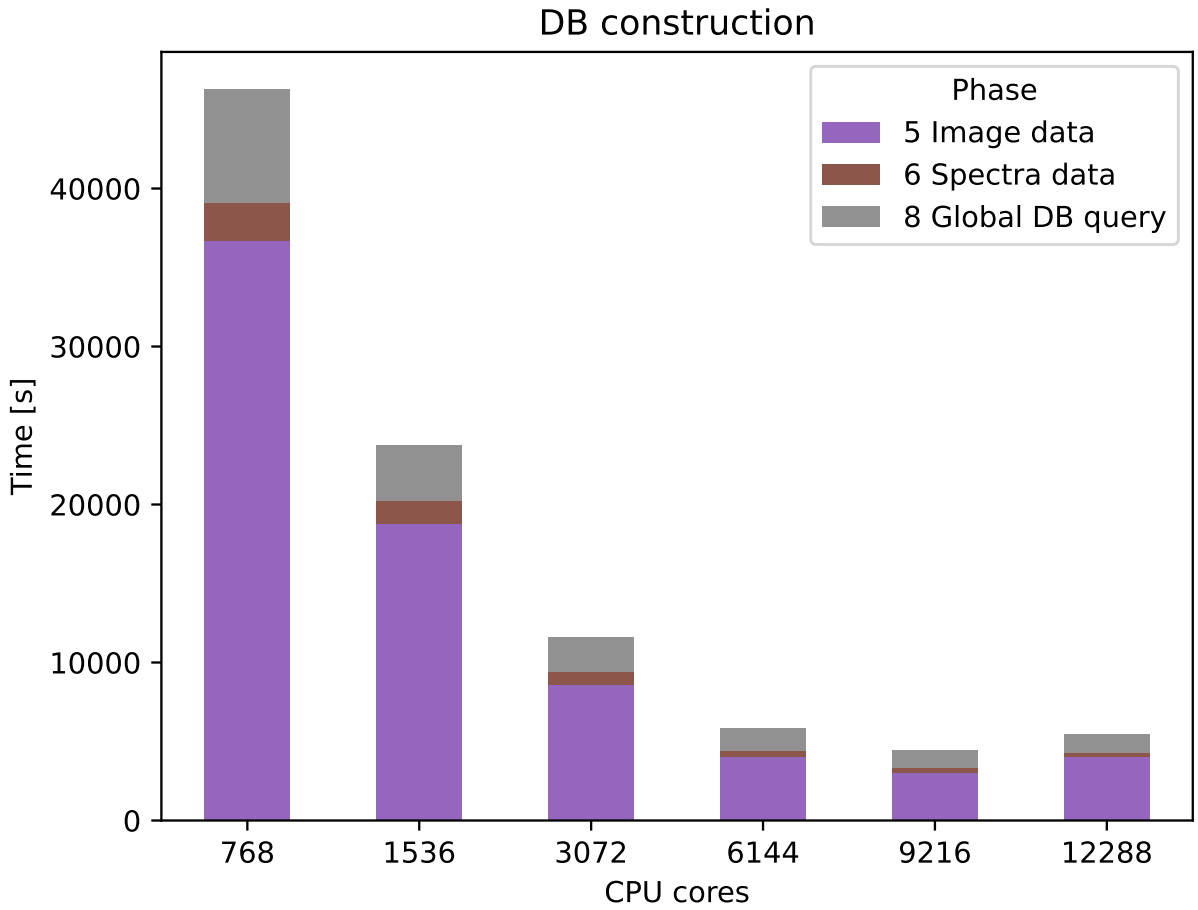

Figure 9.6: The HiSS-Cube running times of the individual phases from Fig. 6.2. These phases are further analyzed from I/O performance point of view.

in the underlying file system, more specifically, the Shared Write I/O bandwidth of shared writes to a single file.

Note that the *Other* part of the overall time spent is not improving with increasing the amount of compute nodes. The reason is that a bigger amount of compute nodes generates higher I/O contention for the write operations, and even though the cumulative time spent in the *Write time* is not higher, the distribution among individual workers is more irregular. For the *Image data* phase it means that for 6144 CPU cores, the slowest worker spent 578 seconds in the *Write time*, but for 12288 CPU cores, the slowest worker spent 667 seconds in the *Write time*, even though he had half of the work to do (both runs processed 1 million images). This results in a higher likelihood of other workers waiting relatively longer for the last worker to finish his batch, because the performance is distributed less evenly. Another aspect is that the higher amount of compute nodes, the bigger the relative processing time of one batch to the overall running time of the application (for our experiments, the batch size is constant for each phase). This might seem to be easily solvable by reducing the

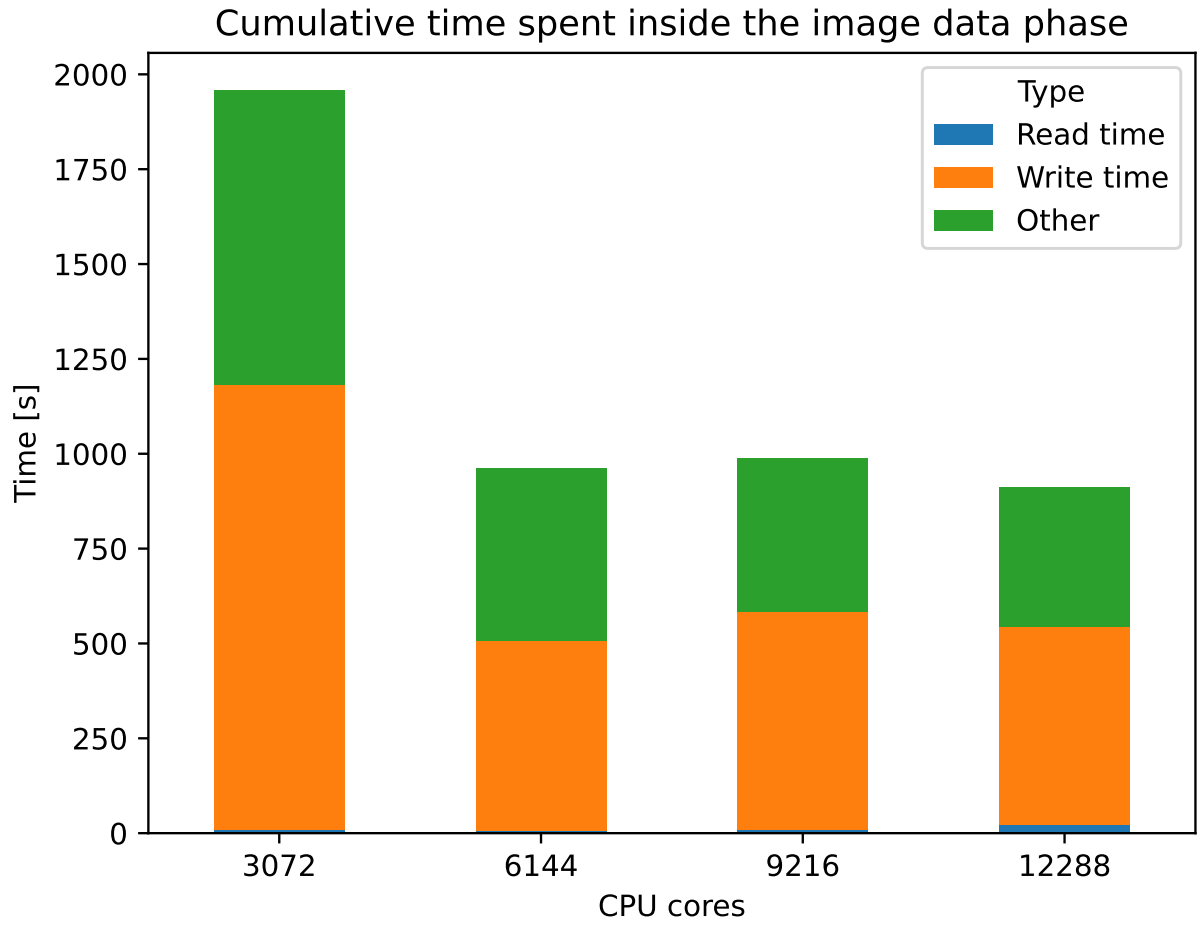

Figure 9.7: Cumulative time spent in I/O functions for *Image data* phase, compared to *Other* running time.

batch size, but in our experience, it resulted in even bigger I/O write contention and more irregular distribution of the I/O resources, leading to higher overall running times.

The good news is that the problem with I/O contention will go away on bigger HPC systems where we will be using a smaller portion of their I/O capacity. Also, the problem with the workers waiting for the last batch will be less severe, because we will be able to afford smaller batch sizes without causing the I/O contention.

Fig. 9.8 shows a similar conclusion. Because the spectra data are much smaller than the image data, the write times still scale up to 12288 CPU cores, keeping the spectra processing more efficient in terms of compute nodes allocated to the job. Eventually, the read operations will become the bottleneck, many small files (FITS spectra files) are accessed in the file system and the time spent in filesystem metadata operations already reaches its limit on 3072 CPU cores (24 compute nodes).

The time spent in *Other* does not improve again, for similar reasons as with the *Image data* phase. The spectra are much smaller than images, but we use much bigger batches

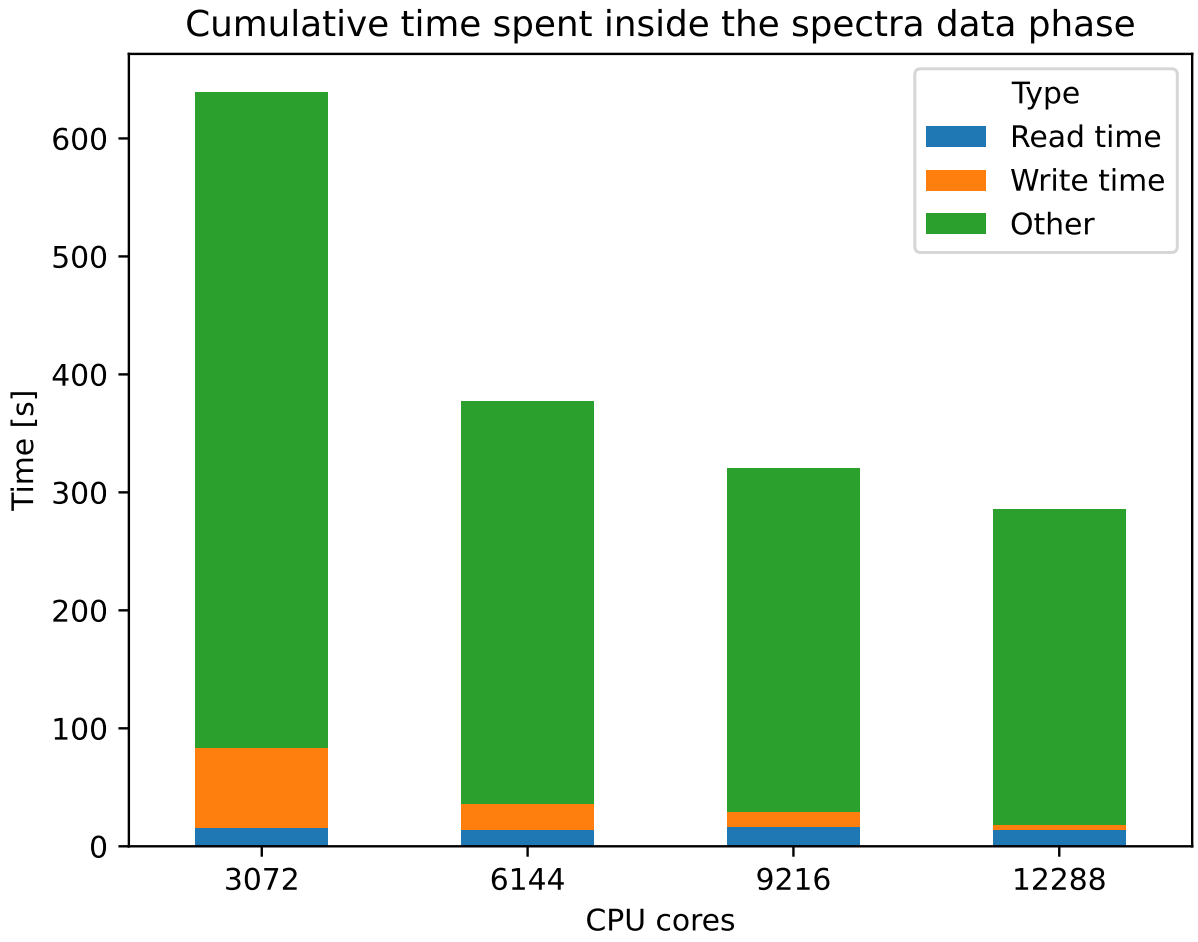

Figure 9.8: Cumulative time spent in I/O functions for *Spectra data* phase, compared to *Other* running time.
.

which results in similar sizes of each write operation.

Fig. 9.9 displays a different pattern. Since the *Parallel database query* follows the links between spectra and their relevant images to combine all the data, it spends a lot of time in read operations. The inefficiency of this phase is much higher compared to other phases if the amount of compute nodes allocated is not convenient for the `Lustre OSTs` count (9216 CPU cores or 72 compute nodes). This phase reaches the performance limit of the underlying file system in terms of read operations on approximately 6144 CPU cores (48 compute nodes), more specifically in the Shared Read I/O bandwidth (the number of I/O operations is not limiting on this filesystem). This is because we use contiguous layouts for the *Stacked_images* datasets from where the data is being read. An option would be to use a chunked layout that would reduce the amount of data that needs to be read within the *Parallel database query*. However, this would slow down the *Image data* phase significantly.

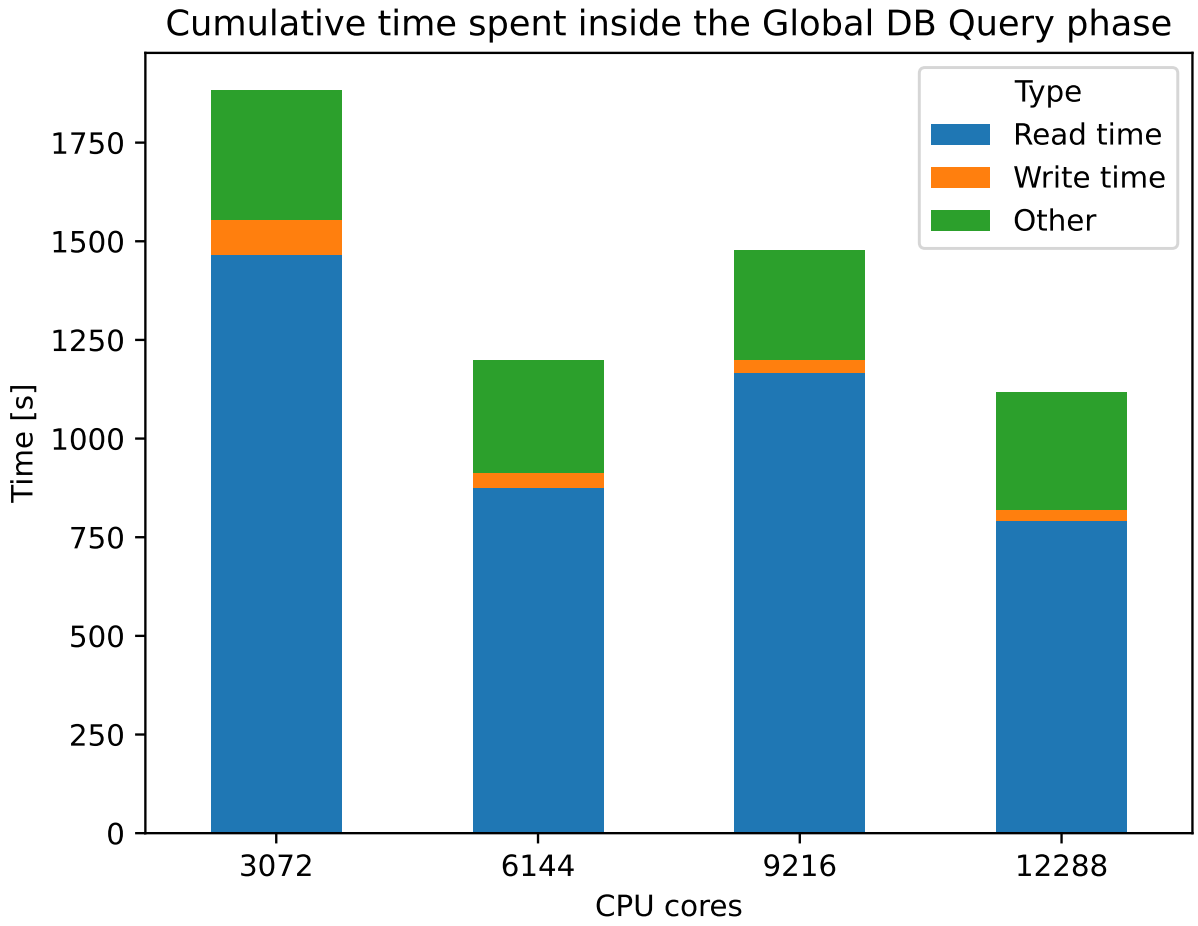

Figure 9.9: Cumulative time spent in I/O functions for *Parallel database query phase*, compared to *Other* running time.

To summarize, the I/O bottlenecks discussed in this section are expected and show that HiSS-Cube behaves efficiently in all phases that do not need to read or write big amounts of data. For these reasons, HiSS-Cube is limited by the performance of the underlying filesystem.

Chapter 10

Conclusion

In this thesis, we designed and developed HiSS-Cube - a generic framework for big dimensionally multi-modal multi-resolution data combination based on HDF5. We have demonstrated its capabilities on astronomical spectra and images and also shown that it is easily extensible to other types of data coming from different detectors. We have solved the hierarchical access as well as providing uncertainties at every resolution. We have shown that the visualization of combined spectra and images in HiSS-Cube works seamlessly with traditional Virtual Observatory tools, as well as native HDF5 readers, such as `H5Web`.

The performance measurements of sequential processing show that the HDF5-based approach is comparable to the FITS-based approach in compression ratios and fifty to five hundred times faster at compressed data, even when the FITS-based algorithm was running in ideal conditions. The key concepts that enabled this efficiency were the region referencing, chunking, and compression filter capabilities of the HDF5.

In addition to that, we have achieved our primary goal of fully parallel implementation of both HiSS-Cube database construction and HiSS-Cube database queries for multi-modal multi-dimensional multi-resolution big data. Our experiments show that HiSS-Cube is now I/O-bound and its performance is limited only by the performance of the underlying parallel file system.

We designed a scalable structure on top of the HDF5 file within HiSS-Cube that enables efficient *Parallel database construction* and *Parallel database queries*. This was verified by combining 5 million spectra and 4.5 million images from the astronomical Sloan Digital Sky Survey. *Parallel database construction* took approximately 60 minutes on our hardware. Combining the data with a *Parallel database query* that retrieves the data for pairs of all spectra and their overlapping image regions finishes in approximately 18 minutes when run on this database.

We have shown that HiSS-Cube functionality can be easily extended for other types of data, such as spectral cubes coming from SKA survey, and that data from more than two detectors can be combined as well.

We have also shown HiSS-Cube flexibility in annotating any type of data (in addition to astronomical data) using the Nexus standard format. Therefore, any HDF5 client,

that understands the Nexus standard, can easily visualize the HiSS-Cube File. The HiSS-Cube database can be reused as a framework for any kind of dimensionally multi-modal data combination and is easily accessible for community development because it remains completely in `Python`.

We also keep track of the latest additions to the HDF5 library, such as the Subfiling Virtual File Driver, which provides even better performance, as shown in the performance measurements.

Because the HiSS-Cube is I/O bound and scales with the amount of `Lustre OSTs`, it will scale to PB-scale data on more powerful HPCs. For example, Frontier HPC[1] will have 1350 `Lustre OSTs` compared to Karolina HPC 48, giving a potential speedup approximately 28. Therefore, HiSS-Cube can process PB-scale data within similar times to what was observed in our experiments, matching the original requirement to scale to LSST data volumes (tens of PBs). Since this means that HiSS-Cube is able to process PBs of data in tens of minutes on such hardware, theoretically it is also able to process SKA volumes of data (hundreds of PBs) but we expect more bottlenecks and unexpected scalability issues (including the parallel file system instability) to be discovered with such data volumes and these would be subject of the future work.

10.1 Future work

In future work, we will extend HiSS-Cube for different scientific data, among which we start with the Earth sciences as they share the need for spherical tesselation indexes with astronomy.

The second objective is to make it more configurable for users to add different types of data. At this moment, adding different types of data is supported by extending and overriding the functionality of existing HiSS-Cube builders which are Python modules. The goal is to enable users to add their type of dimensionally multi-modal data just by configuration, without the need to understand Python code.

The third objective is to test the HiSS-Cube query results with more scientific applications, such as Star Formation Rate estimations via machine learning for astronomical research purposes. This research will particularly benefit from the data combination we have demonstrated, as the estimations can use relevant data from both images and spectra.

The fourth objective is to test the HiSS-Cube with SKA simulated data for data combination of spectra, images, and spectral cubes at once. The problematic part of this objective is to create enough simulated test data (at least tens of PBs), transfer it to an HPC cluster, and then get enough I/O capacity allocated on the cluster to provide relevant test results for such huge amounts of data.

[1] `https://www.olcf.ornl.gov/2021/05/20/olcf-announces-storage-specifications-for-frontier-exascale-system/`

Bibliography

[1] B. Abolfathi et al. The fourteenth data release of the Sloan Digital Sky Survey: First spectroscopic data from the extended baryon oscillation spectroscopic survey and from the second phase of the Apache point observatory galactic evolution experiment. *Astrophys. J. Suppl. Ser.*, 235(2):42, April 2018. doi:10.3847/1538-4365/aa9e8a.

[2] H. Aihara et al. The eighth data release of the Sloan Digital Sky Survey: First data from sdss-iii. *Astrophys. J. Suppl. Ser.*, 193(2):29, April 2011. doi:10.1088/0067-0049/193/2/29.

[3] Sneha B. et al. Analysis of multispectral imaging with the AstroPath platform informs efficacy of PD-1 blockade. *Science 372*, 2021. doi:10.1126/science.aba2609.

[4] P. Baumann. Array databases. In *Encyclopedia of Database Systems*, pages 165–177. Springer, 2018. doi:10.1007/978-1-4614-8265-9_2061.

[5] E. C. Bellm et al. The Zwicky transient facility: System overview, performance, and first results. *Publications of the Astronomical Society of the Pacific*, 131(995):018002, dec 2018. doi:10.1088/1538-3873/aaecbe.

[6] M. R. Blanton et al. Sloan Digital Sky Survey IV: Mapping the milky way, nearby galaxies, and the distant universe. *Astron. J.*, 154(1):28, June 2017. doi:10.3847/1538-3881/aa7567.

[7] F. Bonnarel et al. The ALADIN Interactive Sky Atlas, February 2000. URL: https://arxiv.org/abs/astro-ph/0002109, arXiv:0002109, doi:10.48550/arXiv.astro-ph/0002109.

[8] M. Calabretta, F. Valdes, E. Greisen, and S. L. Allen. Representations of distortions in FITS world coordinate systems. *ASP Conference Series*, 314:551, June 2004. URL: https://adsabs.harvard.edu/pdf/2004ASPC..314..551C.

[9] M. R. Calabretta and E. W. Greisen. Representations of celestial coordinates in FITS. *Astronomy & Astrophysics*, 395(3):1077–1122, 2002. doi:10.1051/0004-6361:20021327.

[10] S. Charlot and M. Longhetti. Nebular emission from star-forming galaxies. *Mon. Not. R. Astron. Soc.*, 323(4):887–903, May 2001. doi:10.1046/j.1365-8711.2001.04260.x.

[11] Gaia Collaboration et al. Gaia Data Release 3: Summary of the content and survey properties, 2022. arXiv:2208.00211, doi:10.48550/arXiv.2208.00211.

[12] A. Comrie, A. Pińska, R. Simmonds, and A. R. Taylor. Development and application of an HDF5 schema for SKA-scale image cube visualization. *Astronomy and Computing*, 32:100389, July 2020. doi:10.1016/j.ascom.2020.100389.

[13] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, et al. Flexible and efficient IR using array databases. *The VLDB Journal*, 17(1):151–168, September 2007. doi:10.1007/s00778-007-0071-0.

[14] P. Cudre-Mauroux et al. A demonstration of SciDB: A science-oriented DBMS. *PVLDB*, 2:1534–1537, 08 2009. doi:10.14778/1687553.1687584.

[15] P. Fernique, T. Boch, T. Donaldson, D. Durand, et al. MOC - HEALPix multi-order coverage map, 2019. URL: https://www.ivoa.net/documents/MOC/.

[16] P. Fernique et al. Hierarchical progressive surveys. *Astronomy & Astrophysics*, 578:A114, June 2015. doi:10.1051/0004-6361/201526075.

[17] Y. Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

[18] J. L. Gardner. Uncertainties in interpolated spectral data. *Journal of Research of the National Institute of Standards and Technology*, 108(1):69, January 2003. doi:10.6028/jres.108.007.

[19] K. M. Gorski et al. HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759–771, April 2005. doi:10.1086/427976.

[20] E. W. Greisen and M. R. Calabretta. Representations of world coordinates in FITS. *Astronomy & Astrophysics*, 395(3):1061–1075, November 2002. doi:10.1051/0004-6361:20021326.

[21] E. W. Greisen, M. R. Calabretta, F. G. Valdes, and S. L. Allen. Representations of spectral coordinates in FITS. *Astronomy & Astrophysics*, 446(2):747–771, 2006. doi:10.1051/0004-6361:20053818.

[22] M. Howison et al. Tuning HDF5 for Lustre File Systems. 9 2010. URL: https://www.osti.gov/biblio/1050648.

[23] ISO/TC 211. Geographic information — Referencing by coordinates. Standard ISO 19111:2019, ISO/TC 211 Geographic information/Geomatics, 2019. URL: https://www.iso.org/standard/74039.html.

[24] Ž. Ivezić et al. LSST: From science drivers to reference design and anticipated data products. *Astrophys. J.*, 873(2):111, March 2019. doi:10.3847/1538-4357/ab042c.

[25] A. Janowczyk, S. Doyle, H. Gilmore, and A. Madabhushi. A resolution adaptive deep hierarchical (RADHicaL) learning scheme applied to nuclear segmentation of digital pathology images. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 6(3):270–276, April 2016. doi:10.1080/21681163.2016.1141063.

[26] T. Jenness. Reimplementing the hierarchical data system using HDF5. *Astronomy and Computing*, 12:221–228, September 2015. doi:10.1016/j.ascom.2015.02.003.

[27] H. Junklewitz, M. R. Bell, M. Selig, and T. A. Enßlin. RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy. *Astronomy & Astrophysics*, 586:A76, January 2016. doi:10.1051/0004-6361/201323094.

[28] R. Kettimuthu et al. Transferring a petabyte in a day. *Future Generation Computer Systems*, 88:191–198, November 2018. doi:10.1016/j.future.2018.05.051.

[29] M. Könnecke et al. The NeXus data format. *J. Appl. Crystallogr.*, 48(Pt 1):301–305, February 2015. doi:10.1107/S1600576714027575.

[30] V. S. Kumar et al. Architectural implications for spatial object association algorithms. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, May 2009. doi:10.1109/ipdps.2009.5161078.

[31] T. Kurth et al. High-Performance I/O: HDF5 for lattice QCD. 2015. doi:10.48550/arXiv.1501.06992.

[32] A. Lamb et al. The Vertica analytic database. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, August 2012. doi:10.14778/2367502.2367518.

[33] H. W. Leung and J. Bovy. Deep learning of multi-element abundances from high-resolution spectroscopic data. *Monthly Notices of the Royal Astronomical Society*, November 2018. doi:10.1093/mnras/sty3217.

[34] H. Li, C. W. Fu, Li Y., and A. Hanson. Visualizing Large-Scale Uncertainty in Astrophysical Data. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1640–1647, November 2007. doi:10.1109/tvcg.2007.70530.

[35] E. Liarou, S. Idreos, S. Manegold, and M. Kersten. MonetDB/DataCell: Online Analytics in a Streaming Column-Store. *Proceedings of the VLDB Endowment*, 5, 08 2012. doi:10.14778/2367502.2367535.

[36] D. J. Maguire. ArcGIS: General purpose GIS software system. In *Encyclopedia of GIS*, pages 25–31. Springer US, 2008. doi:10.1007/978-0-387-35973-1_68.

[37] J. Mainzer et al. Sparse data management in HDF5. In *2019 IEEE/ACM 1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing (XLOOP)*. IEEE, November 2019. doi:10.1109/xloop49562.2019.00009.

[38] S. Maisch, R. Skanberg, and T. Ropinski. A Comparative Study of Mipmapping Techniques for Interactive Volume Visualization. *Journal of WSCG*, 24(2):35–42, 2016.

[39] D. Medvedev, G. Lemson, and M. Rippin. SciServer compute: Bringing analysis close to the data. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management - SSDBM '16*. ACM Press, 2016. doi:10.1145/2949689.2949700.

[40] B. Merín et al. ESA Sky: a new Astronomy Multi-Mission Interface, 2015. arXiv:1512.00842, doi:10.48550/arXiv.1512.00842.

[41] G. Mosby et al. Properties and characteristics of the Nancy Grace Roman Space Telescope H4RG-10 detectors. *J. Astron. Telesc. Instrum. Syst.*, 6(04), November 2020. doi:10.1117/1.JATIS.6.4.046001.

[42] J. Nádvorník. Cross-matching Engine for Incremental Photometric Sky Survey, Diploma Thesis, 2015. URL: https://dspace.cvut.cz/handle/10467/63031?show=full, arXiv:1506.07208.

[43] F. Ochsenbein and R. Williams. VOTable Format Definition Version 1.2. 10 2011. doi:10.5479/ads/bib/2009ivoa.spec.1130o.

[44] P. O'Connor et al. Technology of the LSST focal plane. *Nucl. Instrum. Methods Phys. Res. A*, 582(3):902–909, December 2007. doi:10.1016/j.nima.2007.07.120.

[45] D. Padua et al. MPI-IO. In *Encyclopedia of Parallel Computing*, pages 1191–1199. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_297.

[46] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, November 2016. doi:10.14778/3025111.3025117.

[47] D. C. Price, B. R. Barsdell, and L. J. Greenhill. HDFITS: Porting the FITS data model to HDF5. *Astronomy and Computing*, 12:212–220, September 2015. doi:10.1016/j.ascom.2015.05.001.

[48] M. S. Reis and T. J. Rato. An advanced data-centric multi-granularity platform for industrial data analysis. In *Computer Aided Chemical Engineering*, pages 1225–1230. Elsevier, 2019. doi:10.1016/B978-0-12-818634-3.50205-8.

[49] J. A. Richards and X. Jia. Data Fusion. In *Remote Sensing Digital Image Analysis*, pages 293–312. Springer, Berlin, Heidelberg, 1999. doi:10.1007/978-3-662-03978-6_12.

[50] M. Samland et al. Spectral cube extraction for the VLT/SPHERE IFS: Open-source pipeline with full forward modeling and improved sensitivity. 2022. doi:10.48550/arXiv.2210.06390.

[51] B. Scheers. LSST queries in MonetDB, 2017. URL: https://www.monetdb.org/index.php/blog/lsst-in-monetdb.

[52] P. Singh et al. Hyperspectral remote sensing in precision agriculture: present status, challenges, and future trends. In *Hyperspectral Remote Sensing*, pages 121–146. Elsevier, 2020. doi:10.1016/B978-0-08-102894-0.00009-7.

[53] P. Škoda, P. W. Draper, M. C. Neves, D. Andrešič, and T. Jenness. Spectroscopic analysis in the virtual observatory environment with SPLAT-VO. *Astronomy & Computing*, 7-8:108–120, November 2014. doi:10.1016/j.ascom.2014.06.001.

[54] S. A. Smee et al. The multi-object, fiber-fed spectrographs for the sloan digital sky survey and the baryon oscillation spectroscopic survey. *Astron. J.*, 146(2):32, July 2013. doi:10.1088/0004-6256/146/2/32.

[55] I. Snellen et al. Combining high-dispersion spectroscopy with high contrast imaging: Probing rocky planets around our nearest neighbors. *Astron. Astrophys.*, 576:A59, April 2015. doi:10.1051/0004-6361/201425018.

[56] F. Soboczenski et al. Bayesian Deep Learning for Exoplanet Atmospheric Retrieval, 2018. arXiv:1811.03390.

[57] A. Szalay. From SkyServer to SciServer. *The Annals of the American Academy of Political and Social Science*, 675:202–220, 01 2018. doi:10.1177/0002716217745816.

[58] A. S. Szalay and J. M. Taube. Data-Rich Spatial Profiling of Cancer Tissue: Astronomy Informs Pathology. *Clinical Cancer Research*, 28(16):3417–3424, 08 2022. doi:10.1158/1078-0432.CCR-19-3748.

[59] M. Taghizadeh-Popp et al. SciServer: A science platform for astronomy and beyond. *Astronomy and Computing*, 33:100412, October 2020. doi:10.1016/j.ascom.2020.100412.

[60] M. B. Taylor. TOPCAT & STIL: Starlink Table/VOTable Processing Software. In *Astronomical Data Analysis Software and Systems XIV*, volume 347 of *Astronomical Society of the Pacific Conference Series*, page 29, December 2005.

[61] M. B. Taylor, T. Boch, and J. Taylor. SAMP, the simple application messaging protocol: Letting applications talk to each other. *Astronomy and Computing*, 11:81–90, June 2015. doi:10.1016/j.ascom.2014.12.007.

[62] A. R. van Ballegooij. RAM: A multidimensional array DBMS. In *Current Trends in Database Technology - EDBT 2004 Workshops*, pages 154–165. Springer, 2004. doi:10.1007/978-3-540-30192-9_15.

[63] G. Vane, R. O. Green, T. G. Chrien, H. T. Enmark, et al. The airborne visible/infrared imaging spectrometer (AVIRIS). *Remote Sens. Environ.*, 44(2-3):127–143, May 1993. doi:10.1016/0034-4257(93)90012-M.

[64] D. Vohra. *Apache Parquet*, pages 325–335. 09 2016. doi:10.1007/978-1-4842-2199-0_8.

[65] P. Škoda, O. Podsztavek, and P. Tvrdík. Active deep learning method for the discovery of objects of interest in large spectroscopic surveys. *Astronomy & Astrophysics*, 643:A122, November 2020. doi:10.1051/0004-6361/201936090.

[66] Y. Wang, Y. Lu, C. Qiu, P. Gao, et al. Performance evaluation of an Infiniband-based Lustre parallel file system. *Procedia Environ. Sci.*, 11:316–321, 2011. doi:10.1016/j.proenv.2011.12.050.

[67] D. C. Wells, E. W. Greisen, and R. H. Harten. FITS - a Flexible Image Transport System. *Astronomy & Astrophysics*, 44:363, June 1981.

[68] R. L. White et al. Tiled Image Compression Convention, 2009. URL: http://fits.gsfc.nasa.gov/registry/tilecompression.html.

[69] L. Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques - SIGGRAPH '83*. ACM Press, 1983. doi:10.1145/800059.801126.

[70] R. Wilton, S. J. Wheelan, A. S. Szalay, and S. L. Salzberg. The Terabase Search Engine: a large-scale relational database of short-read sequences. *Bioinformatics*, 35(4):665–670, 07 2018. doi:10.1093/bioinformatics/bty657.

[71] A. Wootten and A. R. Thompson. The Atacama large millimeter/submillimeter array. *Proc. IEEE*, 97(8):1463–1471, August 2009. doi:10.1109/jproc.2009.2020572.

[72] H. Xu. Big data challenges in genomics. In *Handbook of Statistics*, pages 337–348. Elsevier, 2020. doi:10.1016/bs.host.2019.08.002.

[73] R. Yan et al. SDSS-IV/MaNGA: Spectrophotometric calibration technique. *Astron. J.*, 151(1):8, December 2015. doi:10.3847/0004-6256/151/1/8.

[74] Y. Zhang et al. Astronomical Data Processing Using SciQL, an SQL Based Query Language for Array Data. In P. Ballester, D. Egret, and N. P. F. Lorente, editors, *Astronomical Data Analysis Software and Systems XXI*, volume 461 of *Astronomical Society of the Pacific Conference Series*, page 729, September 2012.

[75] Y. Zhang and Y. Zhao. Astronomy in the big data era. *Data Sci. J.*, 14(0):11, May 2015. doi:10.5334/dsj-2015-011.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] J. Nádvorník. HiSS-Cube: A scalable framework for Hierarchical Semi-Sparse Cubes preserving uncertainties. *Astronomy & Computing*. Volume 36, July 2021, doi: 10.1016/j.ascom.2021.100463

[A.2] J. Nádvorník. Hierarchical Semi-Sparse Cubes-parallel framework for storing multi-modal big data in HDF5 *IEEE Access*. October 2023 doi:10.1109/ACCESS.2023.3323897

Remaining Publications of the Author Relevant to the Thesis

[A.3] J. Nádvorník. Cross-matching Engine for Incremental Photometric Sky Survey. *Arxiv*. June 2015, arXiv:1506.07208

The paper has been cited in:

- Dl Bouchefry, K. and de Souza, R. S., *Knowledge Discovery in Big Data from Astronomy and Earth Observation, Chapter 12 - Learning in Big Data: Introduction to Machine Learning*. January 2020, pages 225-249. doi:10.1016/B978-0-12-819154-5.00023-0

[A.4] J. Nádvorník. Time Series Cube Data Model. *Arxiv*, Feb 2017, arXiv:1702.01393

[A.5] J. Nádvorník. HiSS data cubes, HDF5 and the VO. *IVOA Interop*, May 2021, https://wiki.ivoa.net/internal/IVOA/InterOpMay2021Apps/HiSS-cube-HDF5-VO.pdf

[A.6] J. Nádvorník. HDF5 parallelization for Hierarchical Semi-Sparse data cubes. *Astronomical Data Analysis Software and Systems (ADASS) XXXI*. October 2021, https://www.adass2021.ac.za/ADASSXXXI-Posters.pdf

[A.7] J. Nádvorník. The story of Hierarchical Semi-Sparse cubes in HDF5. *HDF5 User Group meeting*. June 2022, https://www.hdfgroup.org/wp-content/uploads/2022/05/Story-of-HiSS-cubes-in-HDF5-HUG-2022.pdf

Appendix A

Database construction for SDSS data

This chapter includes sections of the IPython notebook documentation for the whole preprocessing done by HiSS-Cube as a part of the database construction. It can also be found online as a part of the GitHub code[1]. It focuses on specific steps needed for SDSS spectra and images preprocessing, but similar steps need to be done for any kind of dimensionally multi-modal data to be able to combine them.

Note that we have omitted unnecessary code from the IPython notebook that is needed for the plotting and initialization. We kept only relevant pieces of code that transform or manipulate the data. The purpose is to provide the reader with an understanding of each particular step of the preprocessing described within Sec. 4.1 by visualizing the results of each step. The terms highlighted `like this` refer to the SDSS Data model[2] vocabulary.

[1] `https://github.com/nadvornikjiri/HiSS-Cube/blob/master/notebooks/SDSS_cube.ipynb`
[2] `https://data.sdss.org/datamodel/`

A.1 Extracting uncertainty for image measurements.

In this step, we extract the uncertainty for an SDSS image and visualize all the intermediate by-products below. One further step needs to be done after these, that is conversion to the flux densities used by the spectra. That involves not only conversion to flux but also recalibrating by the image pixel size versus the spectrograph aperture size ratio to make it a density per comparable area.

Steps involved:

1. Reading of `sky background` from the FITS extension.

2. Interpolation of the `sky background` to the image resolution.

3. Reading of `calibration vector` and extend it to the whole image (1D calibration vector works because of the way SDSS continuously reads out the image).

4. Subtraction of the `sky background` and image calibration from the calibrated image, thus producing the image before calibration in `Data Number` units.

5. Calculation of `Errors` in the `Data Numbers`.

6. Conversion of the `Data Number errors` to `Nanomaggies`

7. Calculation of Error-to-signal ratio. It demonstrates what we would expect - for brighter areas of the image we are more certain about the values than for dark areas.

```
def get_image_with_errors(fitsPath):
    with fitsio.FITS(fitsPath) as f:
        fits_header = f[0].read_header()
        img = f[0].read()
        x_size = fits_header["NAXIS1"]
        y_size = fits_header["NAXIS2"]
        camcol = fits_header['CAMCOL']
        run = fits_header['run']
        band = fits_header['filter']
        allsky = f[2]['allsky'].read()[0]
        xinterp = f[2]['xinterp'].read()[0]
        yinterp = f[2]['yinterp'].read()[0]

        gain = float(photometry.get_ccd_gain(camcol, run, band))
        darkVariance = float(photometry.get_dark_variance(camcol, run, band))

        grid_x, grid_y = np.meshgrid(xinterp, yinterp, copy=False)

        #interpolating sky background from 256x196 small image included in SDSS
        simg = ndimage.map_coordinates(allsky, (grid_y, grid_x),
                                                                    order = 1,
                                                                        mode="nearest")


        calib = f[1].read()

        cimg = np.tile(calib, (y_size, 1)) #calibration image constructed from
            calibration vector

        dn = img / cimg + simg #data numbers detected originally by the detector

        dn_err = np.sqrt(dn/gain + darkVariance) #errors in data numbers (sigma)

        img_err= dn_err*cimg #errors in nanomaggies (sigma)

        img_phys_flux = img

        err_to_signal = img_err / img

        return img, img_err, allsky, simg, cimg, dn, nelec, dn_err, err_to_signal
```

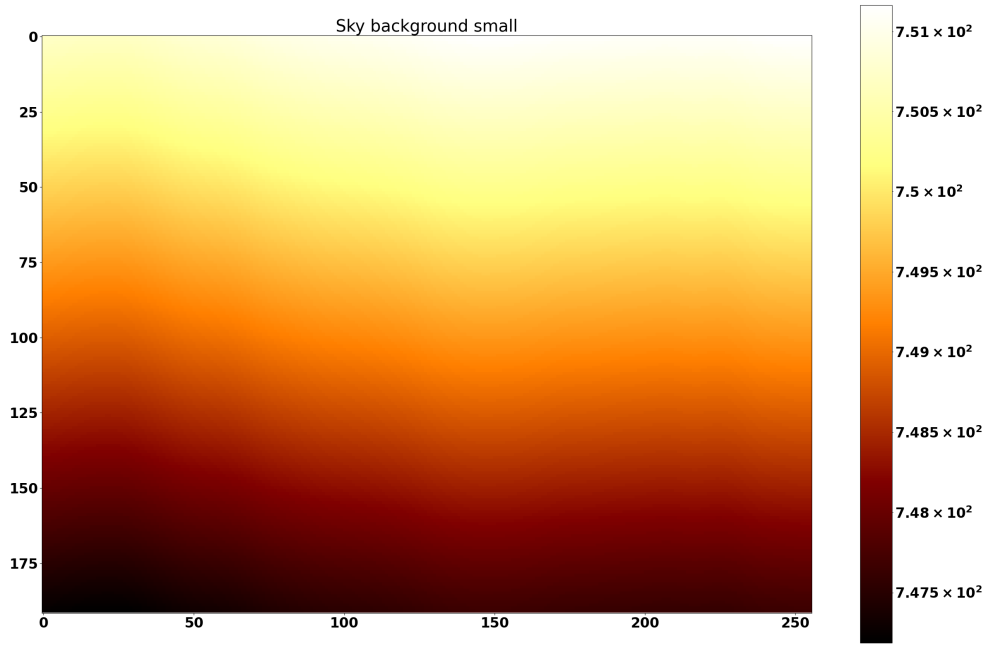

Figure A.1: This image shows the `Sky background` in low resolution. This sky background needs to be subtracted from all pixels of the image to be able to see the magnitudes of astronomical objects behind the atmosphere.

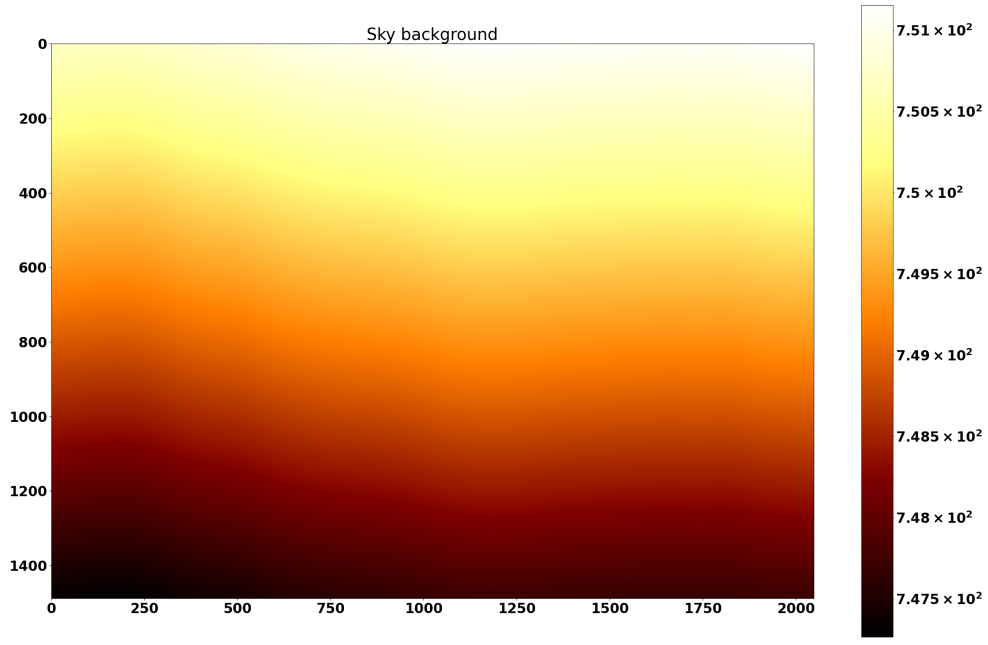

Figure A.2: This image shows the `Sky background` interpolated to the same resolution as the calibrated image.

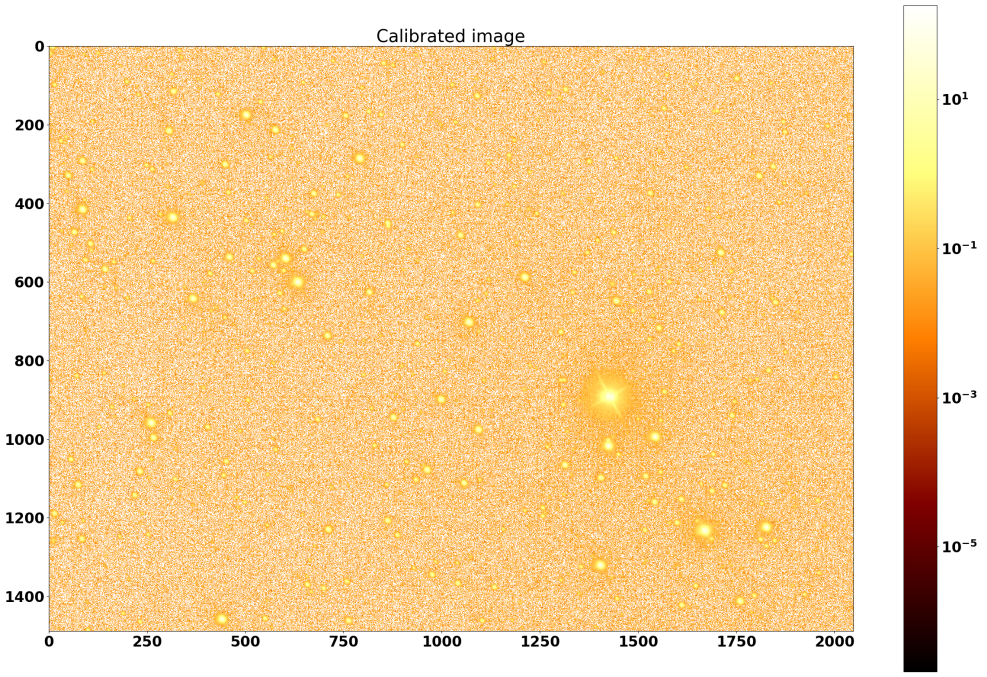

Figure A.3: This is the final image produced by the SDSS pipeline with science-ready values. Each pixel represents the mean light intensity, no errors are included and need to be reconstructed.

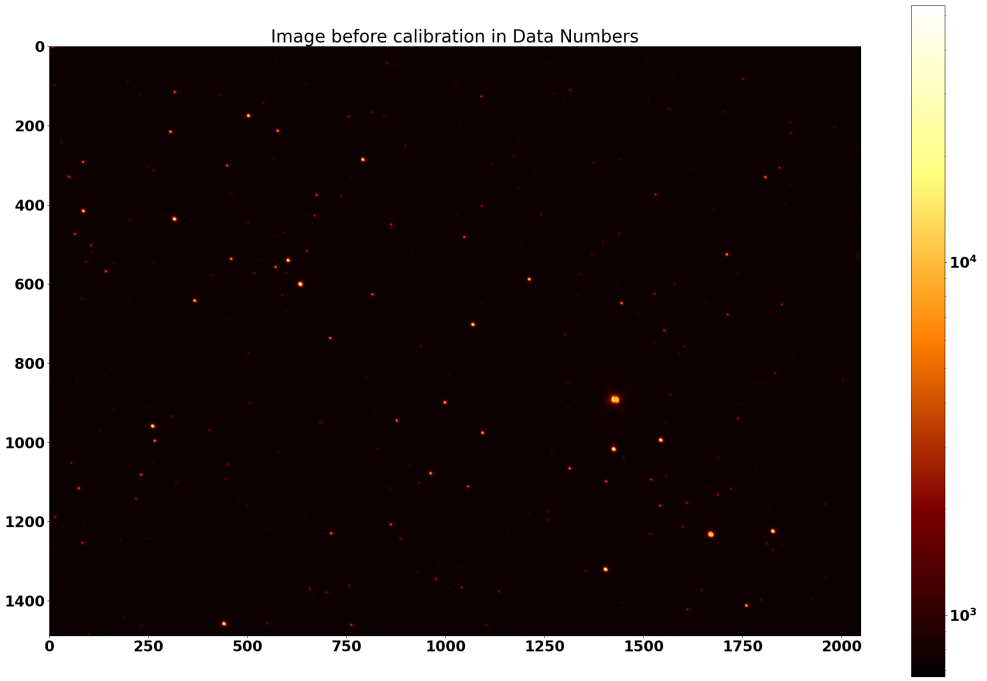

Figure A.4: This is the image after we subtract the `Sky background` and the `Calibration vector`. Therefore, it is uncalibrated image in the `Data Numbers` that are coming from the detector.

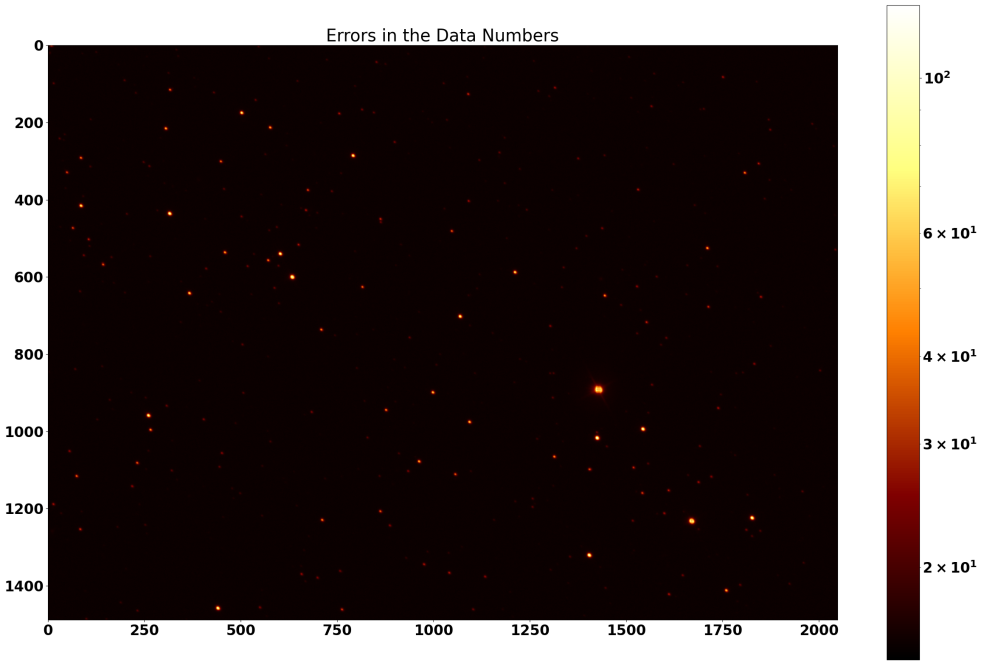

Figure A.5: This image shows the `Errors` reconstructed from the calibrated image in `Data Numbers`. The reconstruction is done from the `gain` of the CCD chip within this `Camera Column` and the `Dark Variance` of the CCD chip, essentially the electronic noise of the chip.

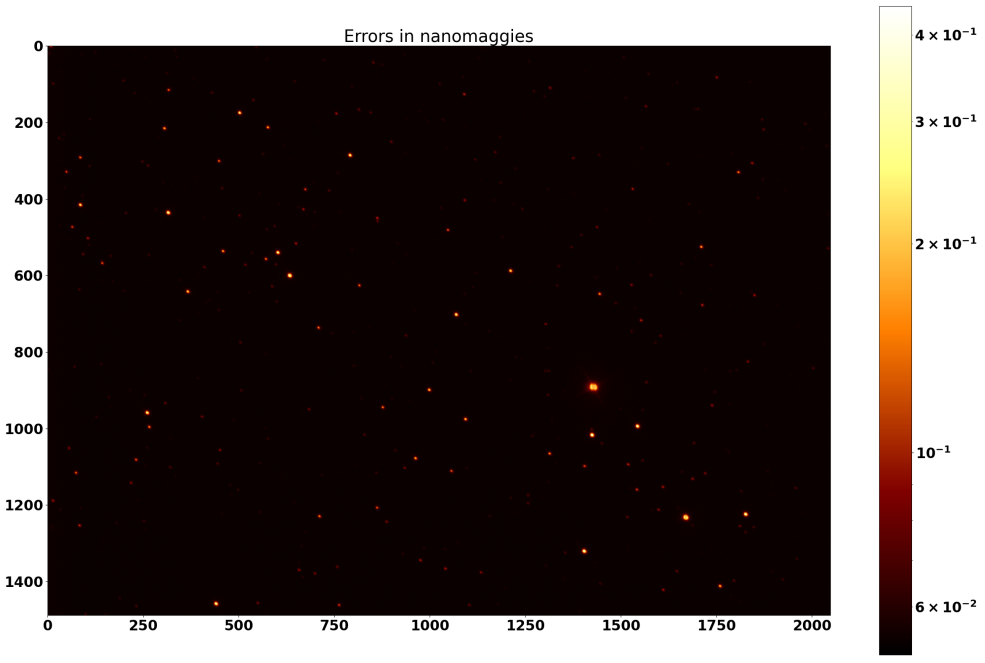

Figure A.6: This is the same image as *Errors in Data Numbers* but converted to `Nanomaggie` unit.

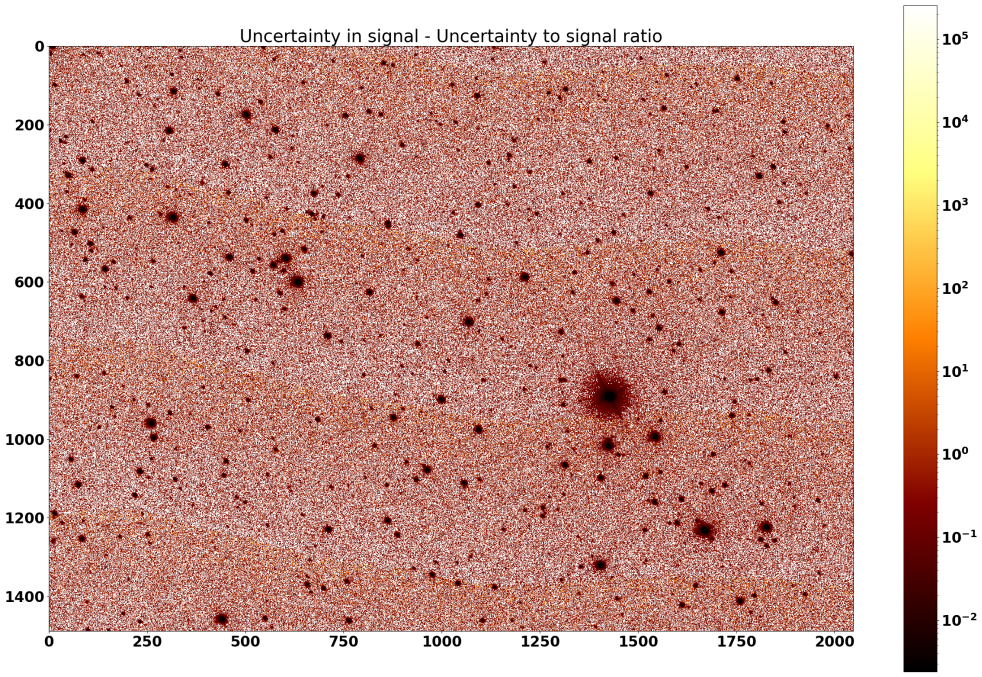

Figure A.7: This image shows the ratio between the `Error` and the calibrated image, therefore, showing the uncertainty compared to the signal, telling how much is the pixel value on the calibrated image uncertain. Brighter areas are more uncertain compared to the signal strength.

A.2 Extraction of spectrum errors

For the spectra, the inverse variance of the measurement is already stored in the FITS files. It is already in flux density ('1E-17 erg/cm^2/s/Ang'), so there is no need to convert it. However, the norm is to store the sigma as inverse variance, therefore we need to invert it to get the variance itself.

```
with fitsio.FITS(fits_path) as hdul:
    data = hdul[1].read()
    flux = data["flux"] * 1e-17
    sigma = np.sqrt(np.divide(1,data["ivar"])) * 1e-17
    wl = np.power(10, data["loglam"])
```

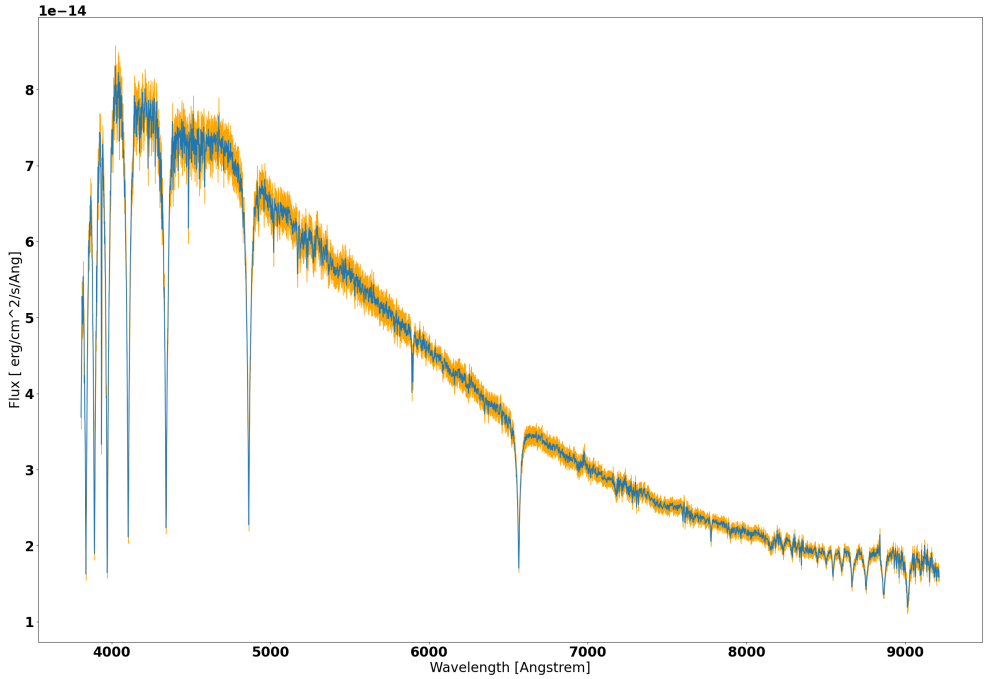

Figure A.8: This plot shows light intensity in different wavelengths. The unit used i flux density.

A.3 Application of Filter transmission curves

We are not able to compare directly spectroscopic measurements with photometric ones. However, we can make the spectra comparable by applying the `Filter transmission curves` to them and thus simulate values that would be measured to photon counts reduced by filter transmission function.

The filter transmission curve is displayed in Fig. A.9 and Fig. A.10. We take the transmission curves for individual filters and take the maximum transmission ratio for each wavelength, which essentially gets rid of the filter overlaps. We can do this because the image coordinates are the mid_points of those filters which are not in the overlapping regions anyway, so there will be no ambiguity in which filter should be applied in those regions.

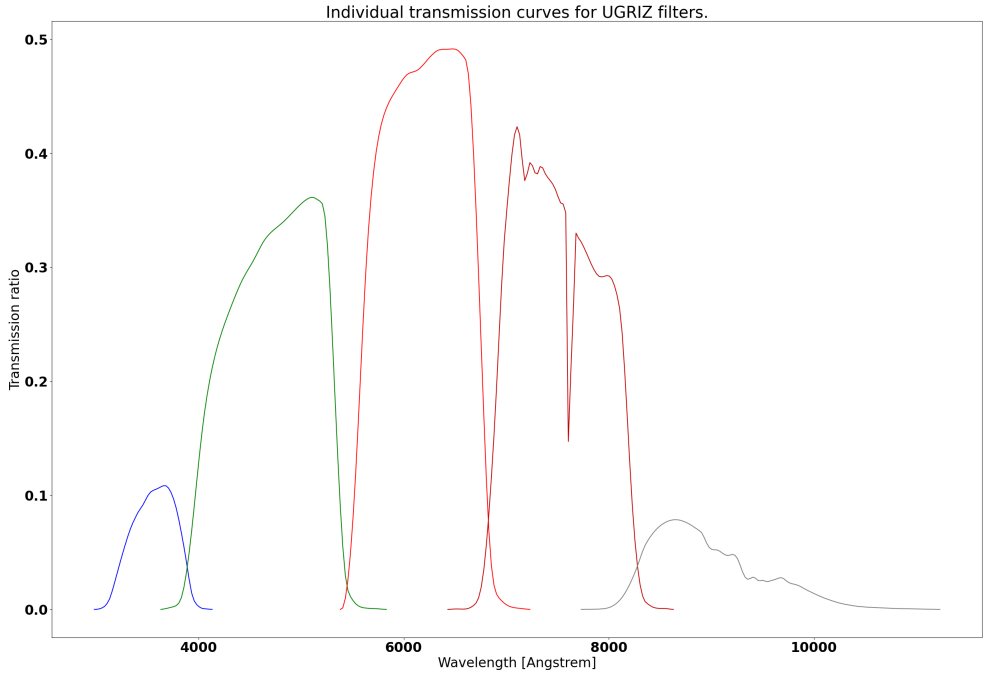

Figure A.9: These curves essentially simulate how much light can get through the individual color filters in the SDSS photometric survey.

```
u = photometry.transmission_curves["u"]
g = photometry.transmission_curves["g"]
r = photometry.transmission_curves["r"]
i = photometry.transmission_curves["i"]
z = photometry.transmission_curves["z"]

merged_transmission_curve = photometry.merged_transmission_curve

wl, band_ratio = zip(*list(merged_transmission_curve.items()))
band, ratio = zip(*list(band_ratio))
u_wl, u_ratio = zip(*list(u.items()))
g_wl, g_ratio = zip(*list(g.items()))
r_wl, r_ratio = zip(*list(r.items()))
i_wl, i_ratio = zip(*list(i.items()))
z_wl, z_ratio = zip(*list(z.items()))
```

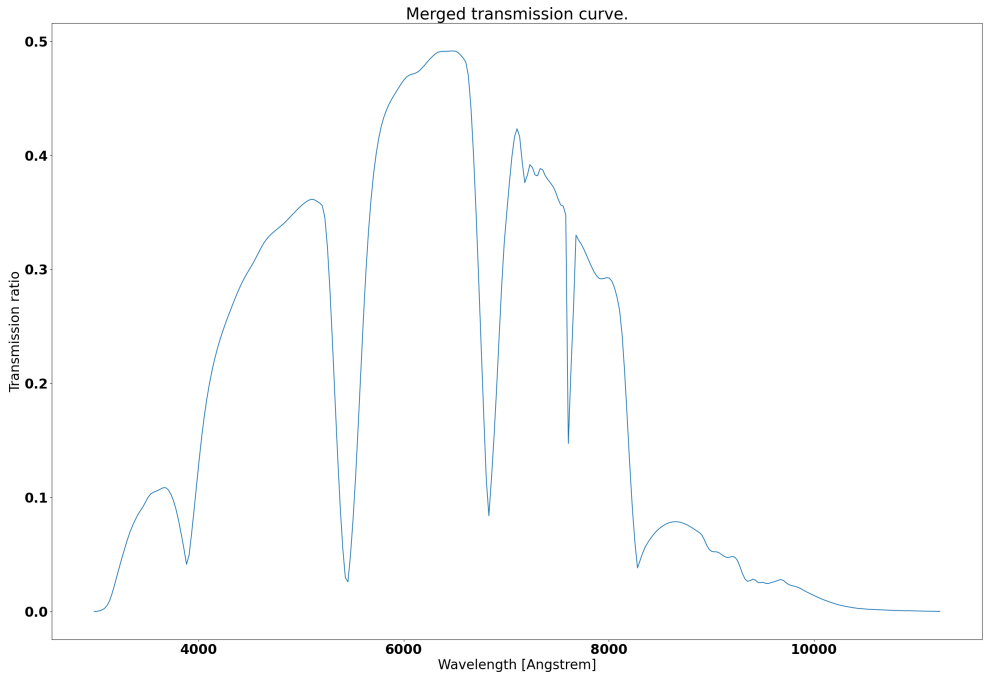

Figure A.10: This artificial construct can be used to simulate light traveling through five distinct photometric filters in different wavelengths throughout all wavelengths used by the SDSS spectra.

A.4 Application of transmission curve

The Fig. A.11 displays the spectrum with the merged photometric transmission curve applied. The magnitude is multiplied in each wavelength by the transmission curve ratio in that wavelength.

```
transmission_ratio, zero_point, softening =
    photometry.get_photometry_params(spec_flux, spec_wl)

photometric_observed_spectrum_flux = spec_flux * transmission_ratio
photometric_observed_spectrum_flux_sigma = spec_sigma * transmission_ratio
```

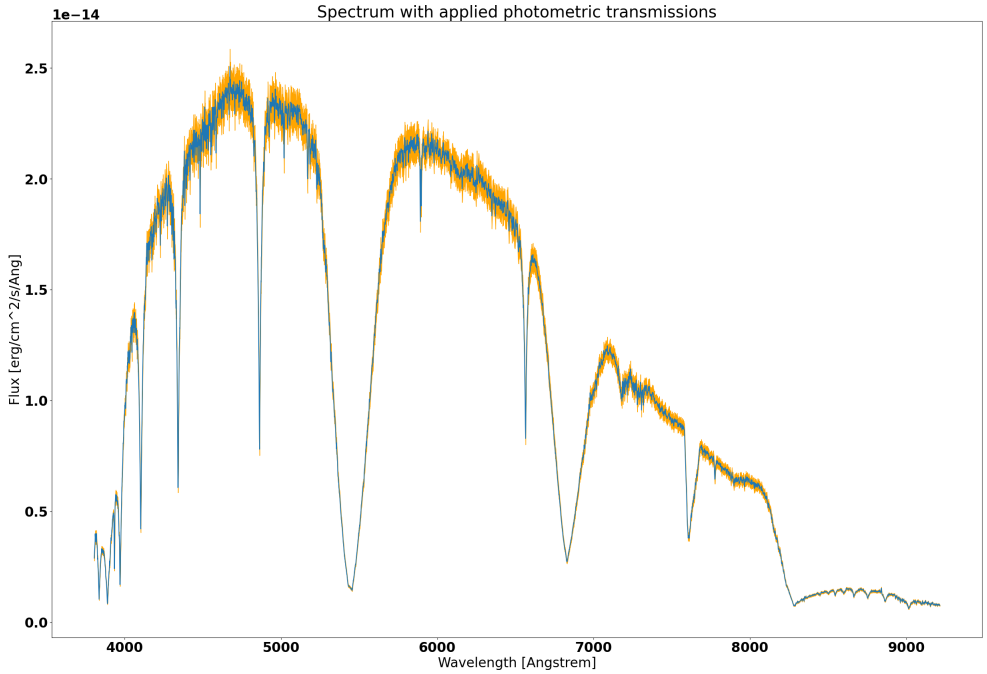

Figure A.11: Spectrum with applied photometric transmission curve.

A.5 Spectrum transformation - rebinning

In this section, we show how the lower resolutions and rebinning is applied to spectra. The rebinning is often needed by machine learning algorithms which need homogeneous input vectors. It is also helpful for the HiSS-Cube database construction in parallel because it lowers the overall number of datasets. Figures A.12 to A.23 show pairs of down-sampled resolutions of spectra, for non-binned and binned variants.

```
spec_zoom_cnt = 2

fits_header, spec_cube =
    photometry.get_multiple_resolution_spectrum(test_spectrum, spec_zoom_cnt,
                                        apply_rebin=False, rebin_min=0, rebin_max=0,
                                        rebin_samples=0, apply_transmission=False)
fits_header, spec_cube_binned =
    photometry.get_multiple_resolution_spectrum(test_spectrum, spec_zoom_cnt,
                                        apply_rebin=True, rebin_min=3839.7244,
                                            rebin_max=8914.597,
                                        rebin_samples=4620,
                                            apply_transmission=False)
```

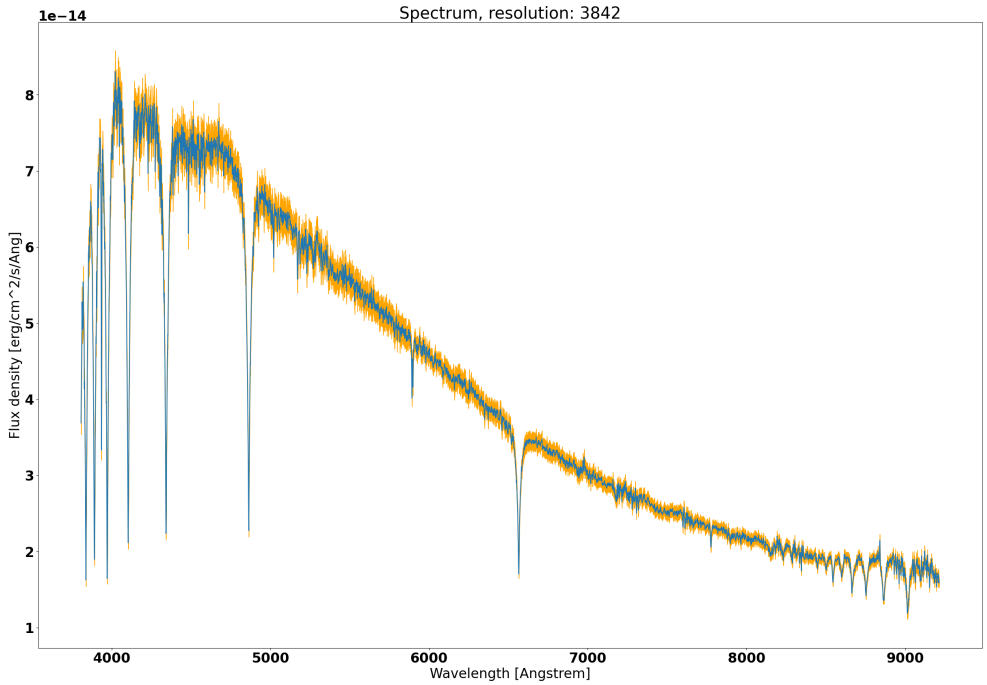

Figure A.12: This is the original spectrum with 3842 points measured at different wavelengths.

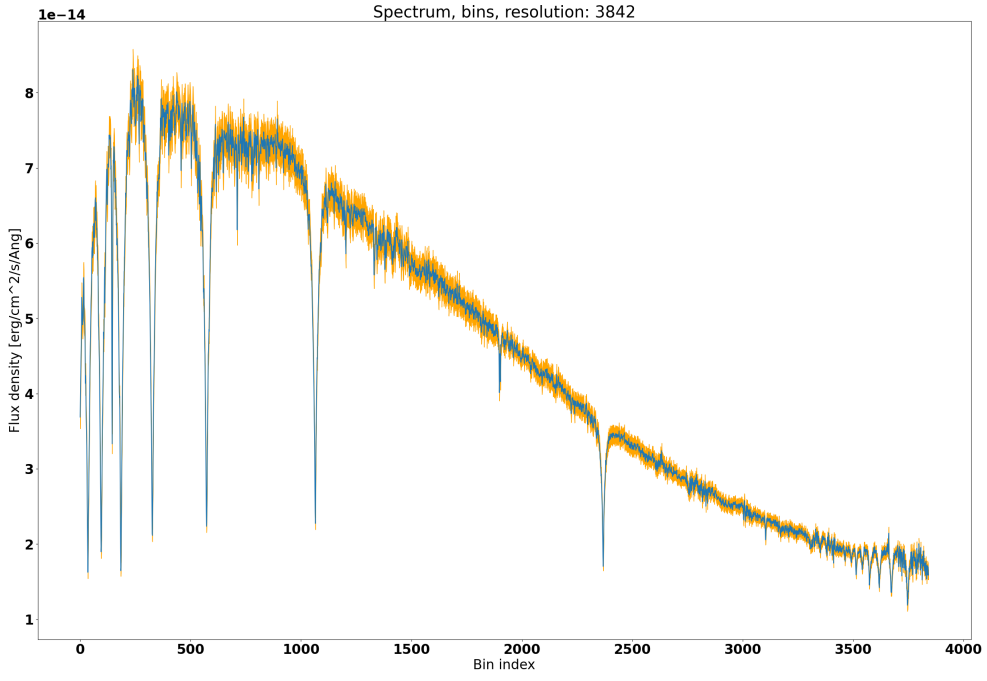

Figure A.13: This is the same plot as Fig. A.12 but with bin index on x axis to show the resolution.

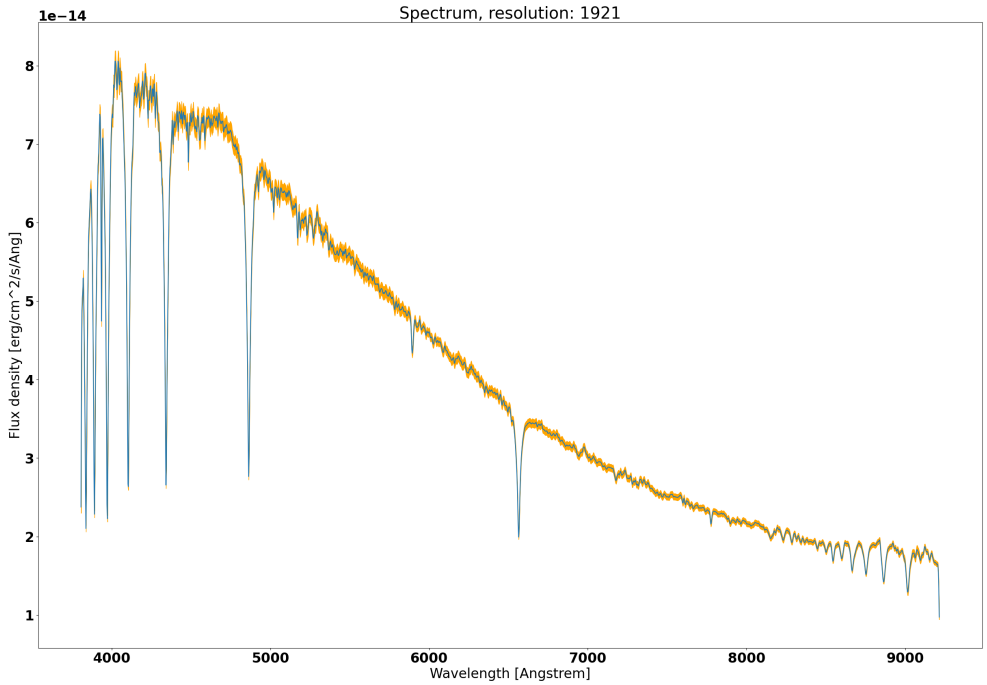

Figure A.14: This is a down-sampled spectrum interpolated via cubic interpolation from the original one at half resolution (every other wavelength of the original).

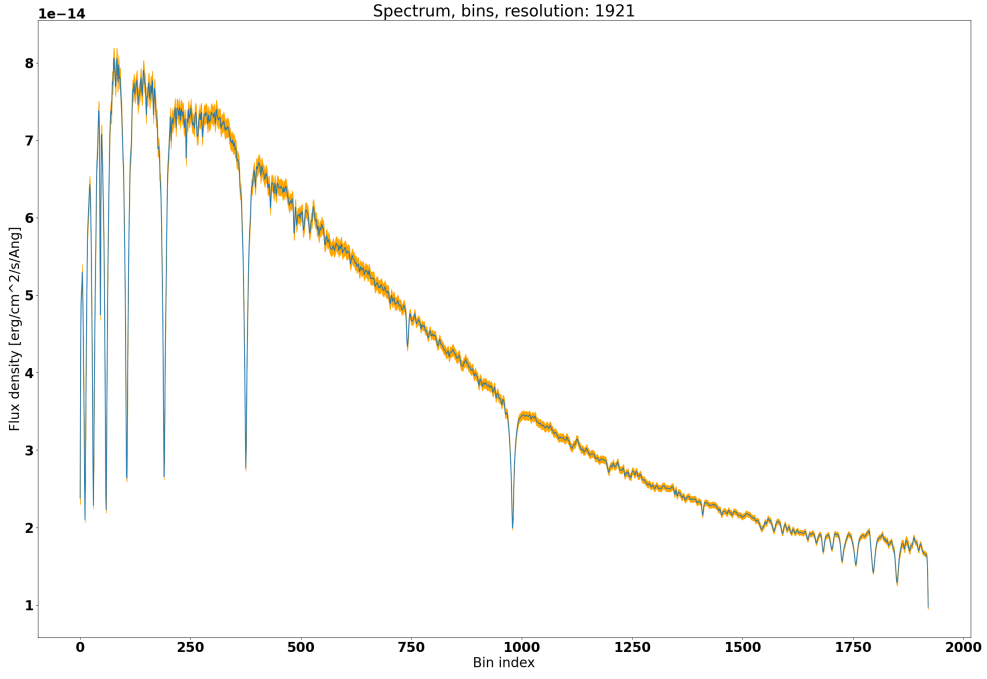

Figure A.15: This is the same plot as Fig. A.14 but with bin index on x axis to show the resolution.

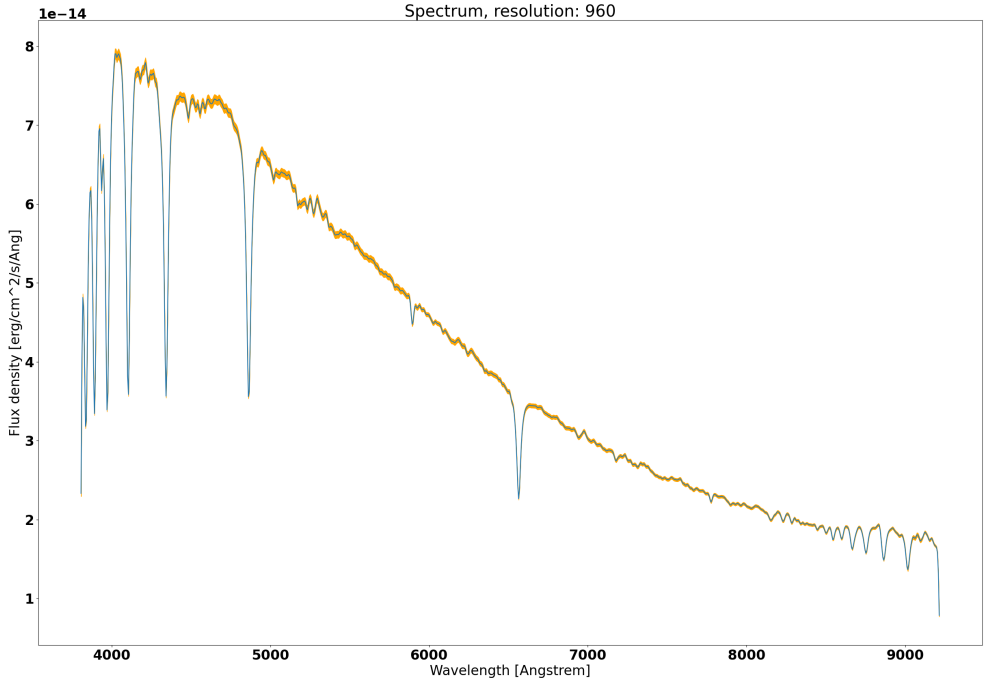

Figure A.16: This is the same plot as Fig. A.14 but with halved resolution.

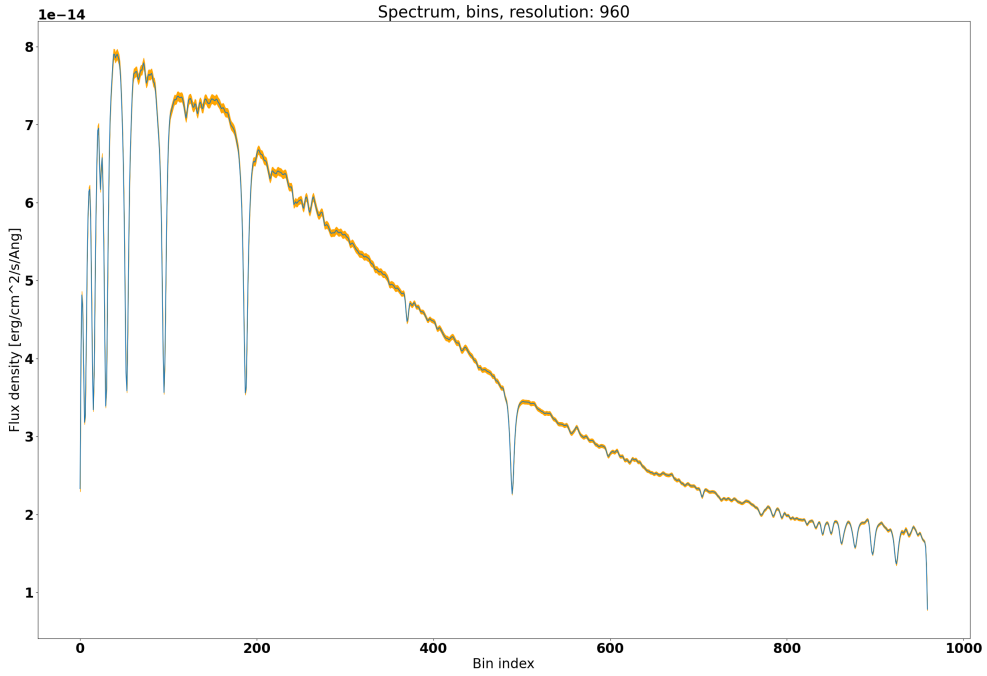

Figure A.17: This is the same plot as Fig. A.16 but with bin index on x axis to show the resolution.

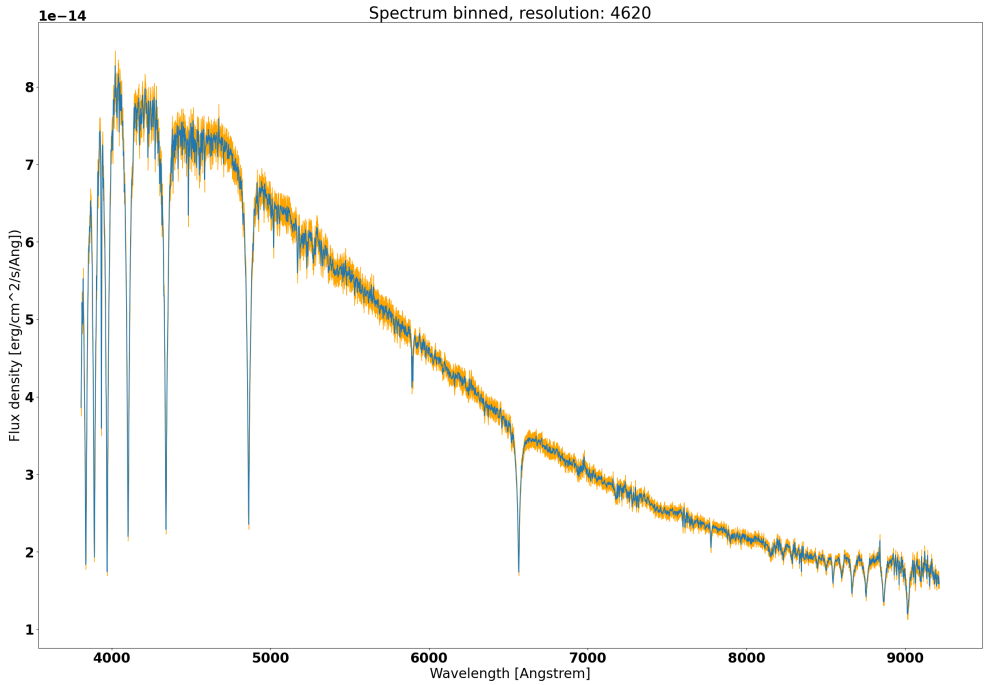

Figure A.18: This is the original spectrum but interpolated onto bins from 3650 to 10400 Angstroms. This is the maximum coverage of the BOSS spectrograph.

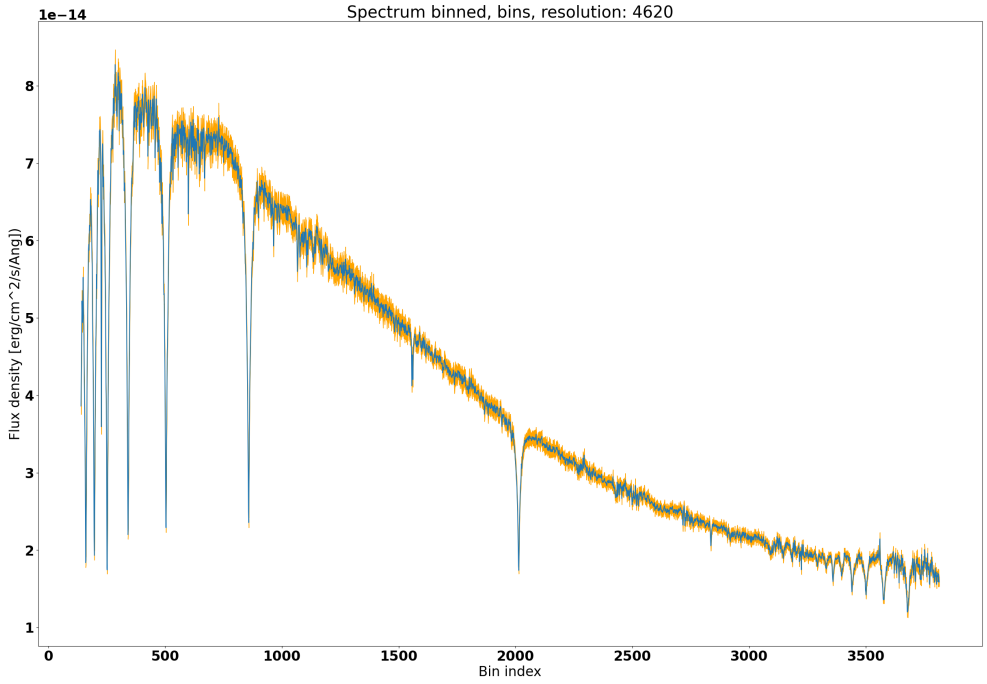

Figure A.19: This is the same plot as Fig. A.18 but with bin index on x axis to show the resolution.

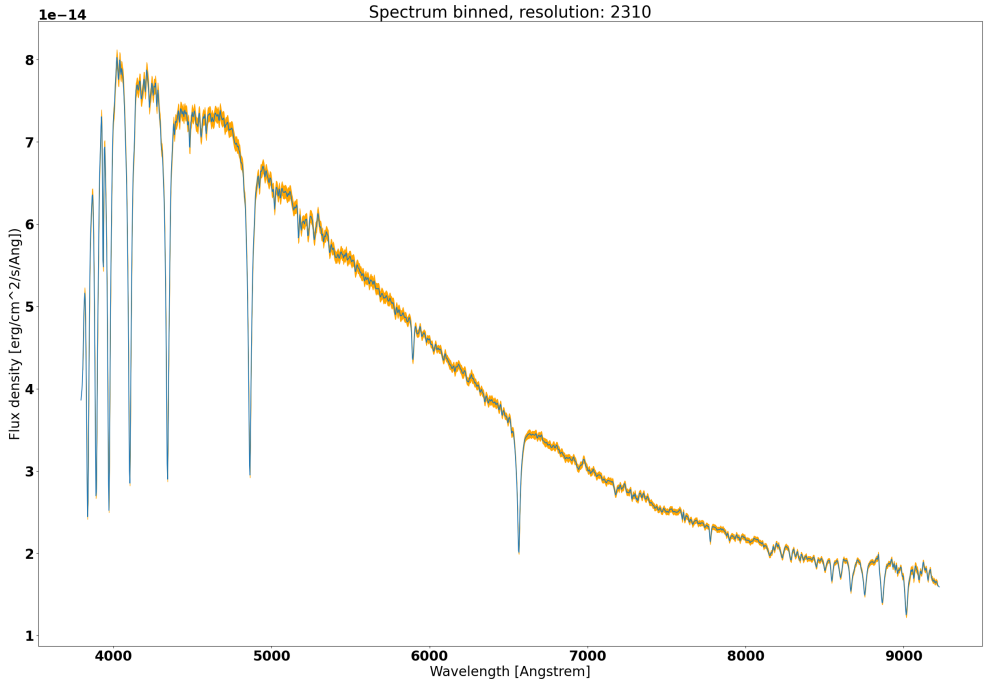

Figure A.20: This is the same plot as Fig. A.18 but with halved resolution.

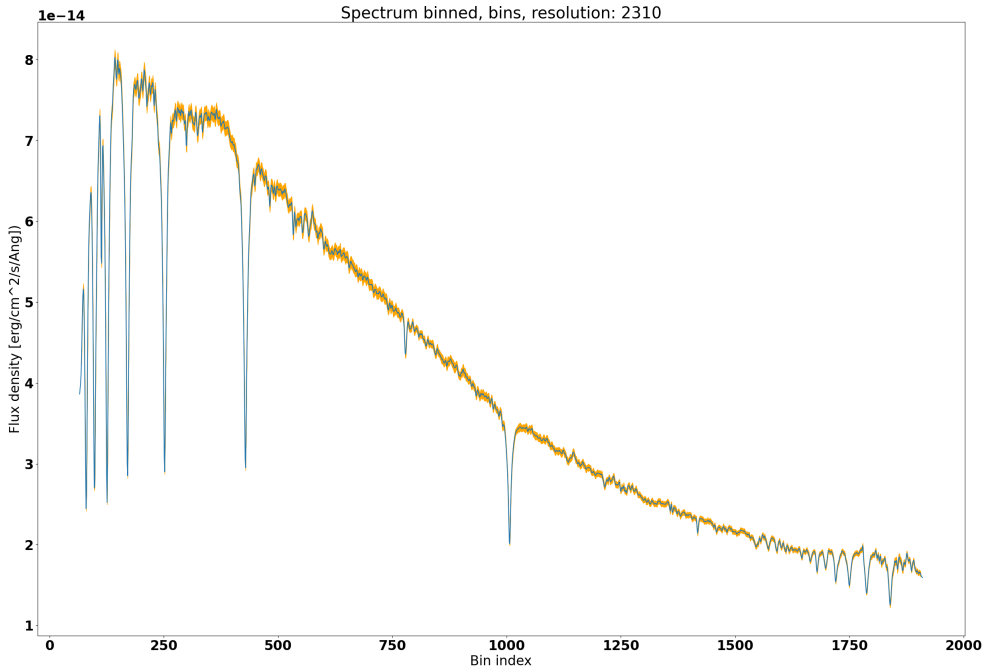

Figure A.21: This is the same plot as Fig. A.20 but with bin index on x axis to show the resolution.

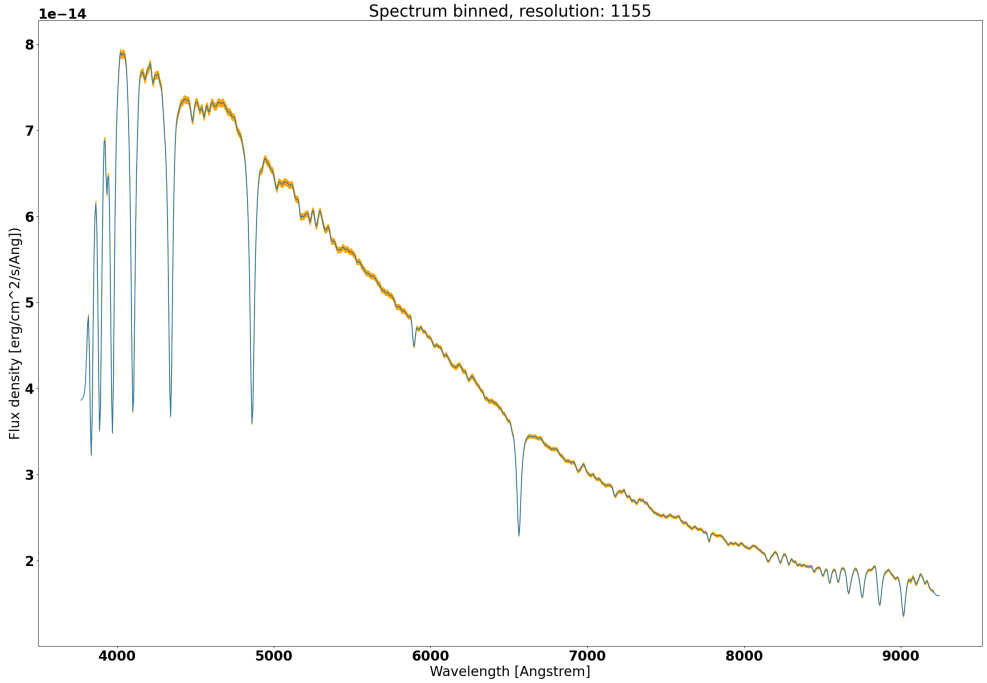

Figure A.22: This is the same plot as Fig. A.20 but with halved resolution.

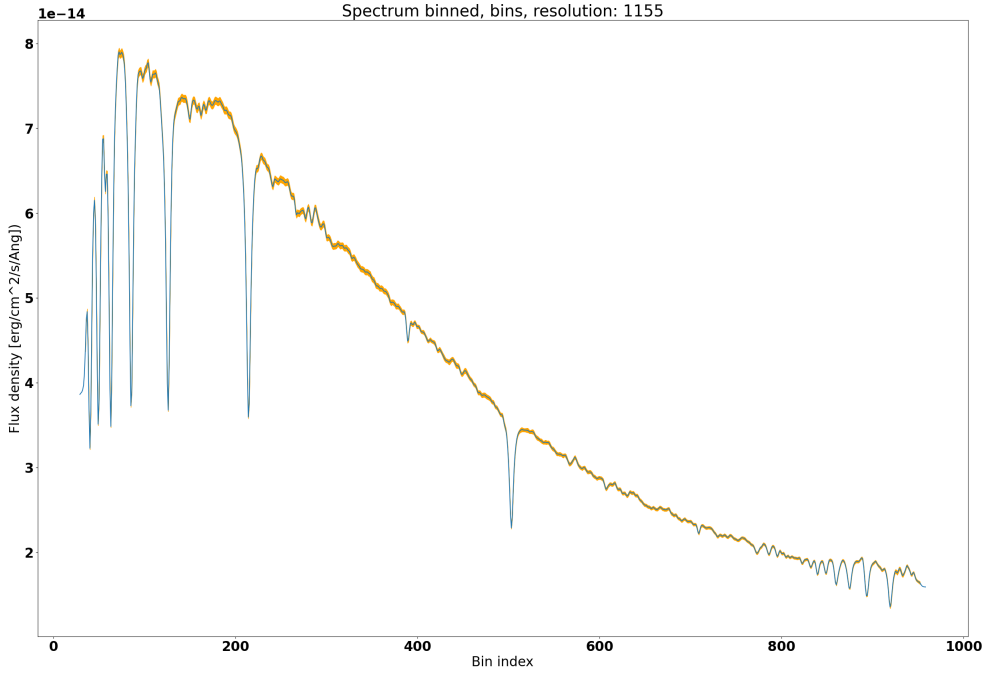

Figure A.23: This is the same plot as Fig. A.20 but with bin index on x axis to show the resolution.

A.6 Spectrum transformation - transmission curves

This section describes how multiple resolution spectrum is constructed, with the same algorithm as above but transmission curves applied. This time there are visualizations of two options.

1. Flux densities without photometric transmission applied
2. Flux densities with photometric transmission applied

For each of these, we construct the down-sampled resolutions down to the MIN_RES setting (Each lower resolution is half of the upper one, meaning we get spectral resolutions e.g. (3842, 1921, 960, etc.)

```
spec_zoom_cnt = 4

fits_header, spec_cube =
    photometry.get_multiple_resolution_spectrum(test_spectrum,
                                                spec_zoom_cnt,
                                                apply_transmission=False)
fits_header, spec_cube_with_transmission =
    photometry.get_multiple_resolution_spectrum(test_spectrum,
                                                 spec_zoom_cnt,
                                                 apply_transmission=True)
```

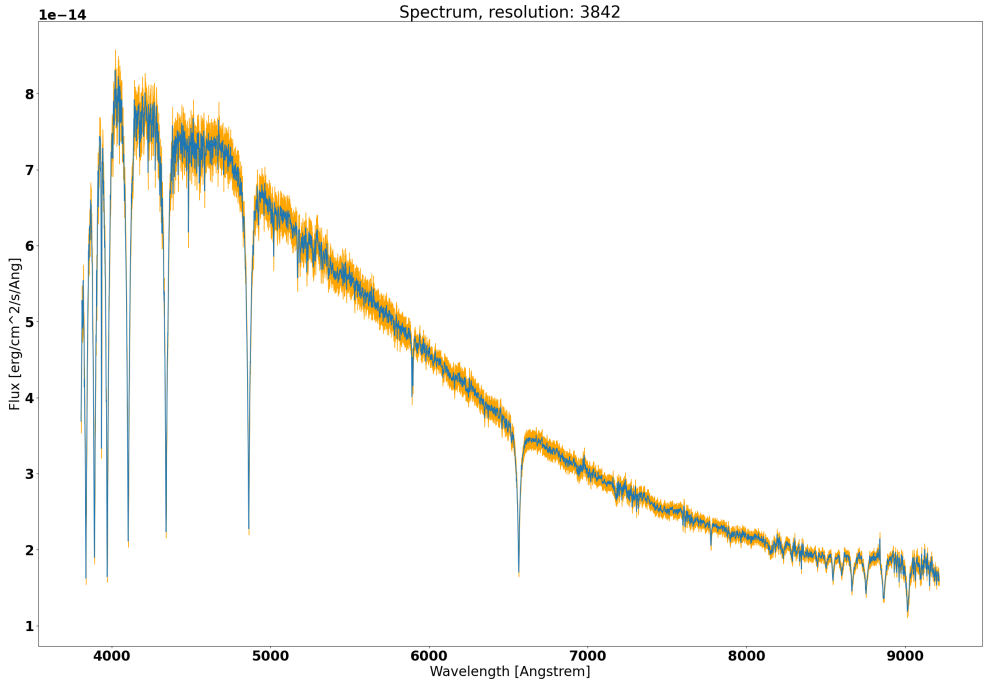

Figure A.24: This is the original spectrum with a resolution of 3842 measured light intensities in different wavelengths.

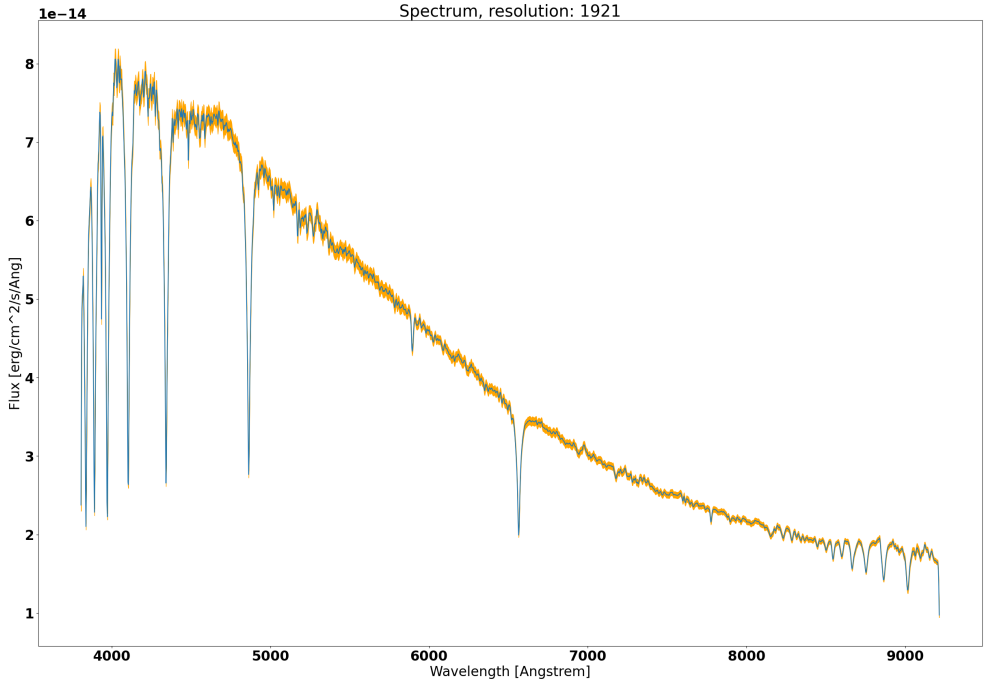

Figure A.25: This is the same plot as Fig. A.24, with the wavelength resolution halved.

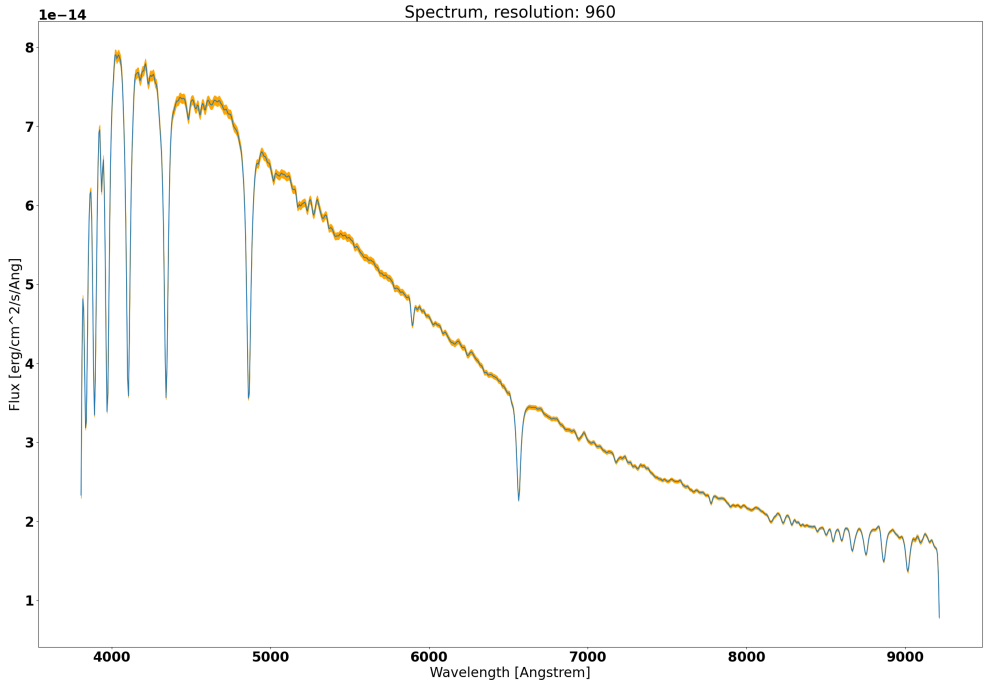

Figure A.26: This is the same plot as Fig. A.25, with the wavelength resolution halved.

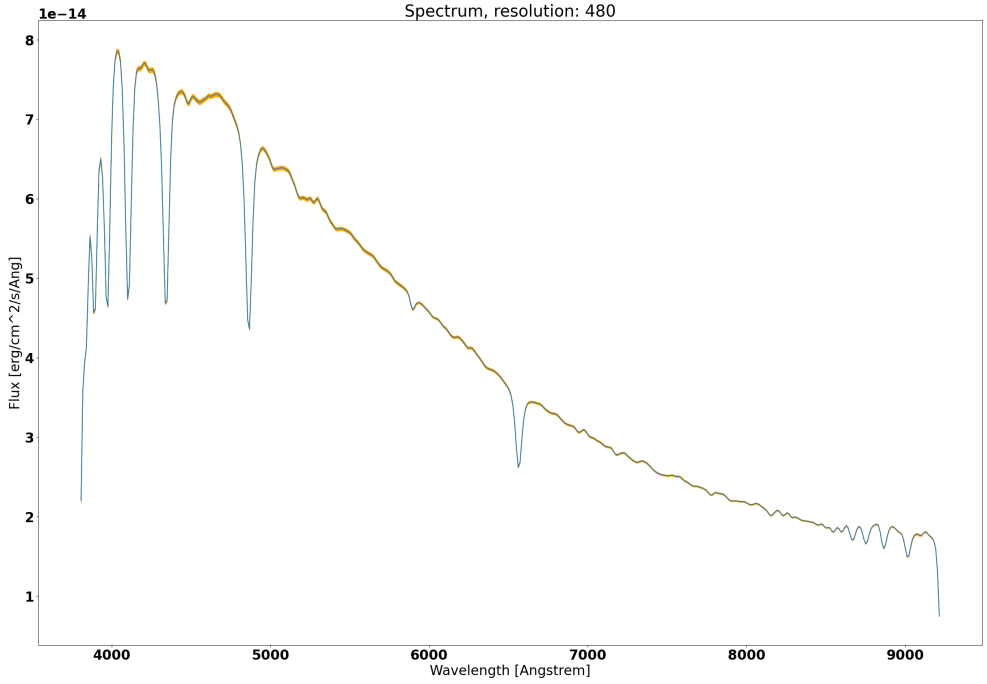

Figure A.27: This is the same plot as Fig. A.26, with the wavelength resolution halved.

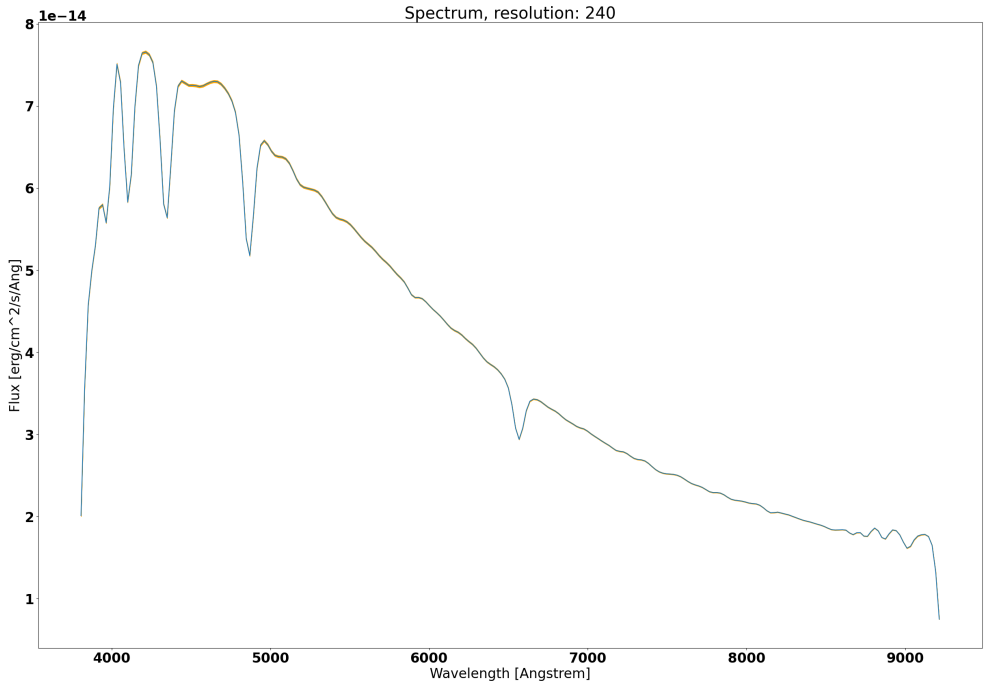

Figure A.28: This is the same plot as Fig. A.27, with the wavelength resolution halved.

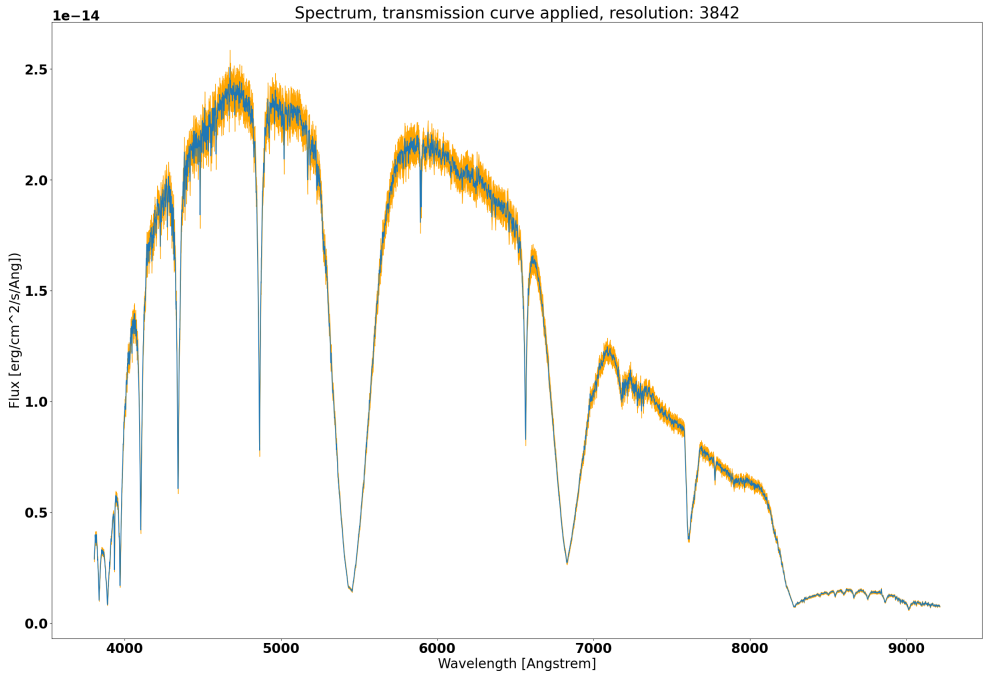

Figure A.29: This is the spectrum with applied transmission curve from Fig. A.10. Every light intensity at a given wavelength is multiplied by the transmission ratio between 0 and 1, depending on how much the photometric filter would transmit the light.

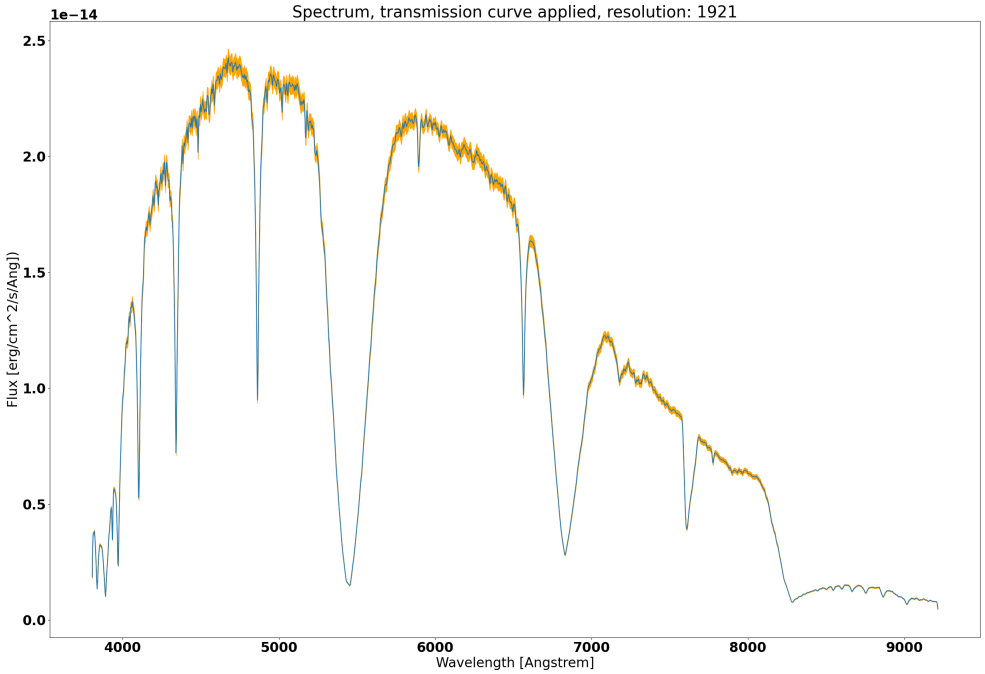

Figure A.30: This is the same plot as Fig. A.29, with the wavelength resolution halved.

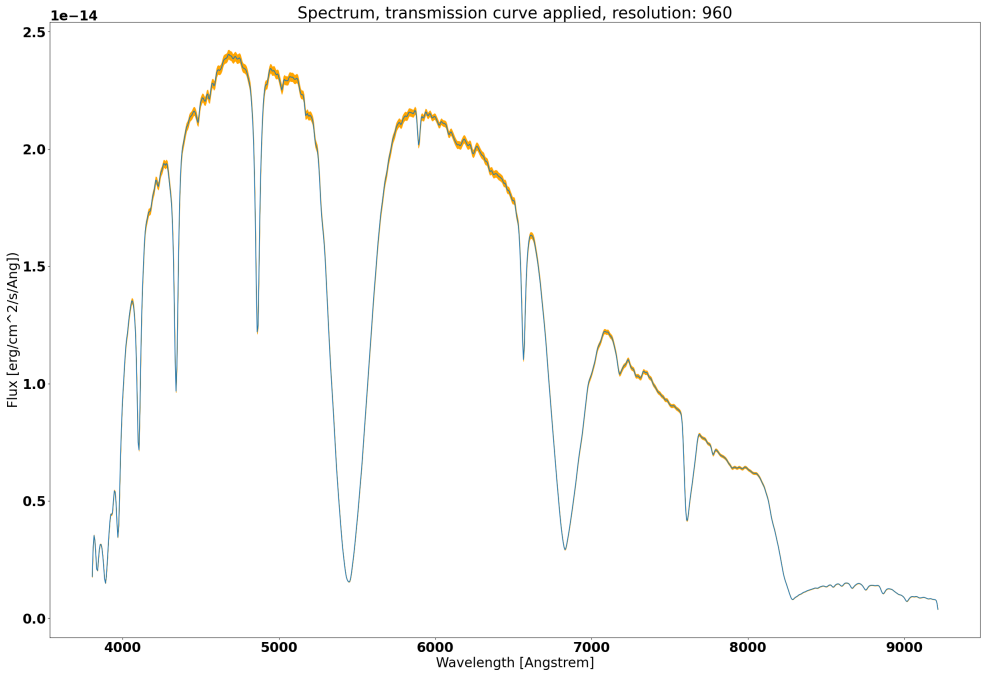

Figure A.31: This is the same plot as Fig. A.30, with the wavelength resolution halved.

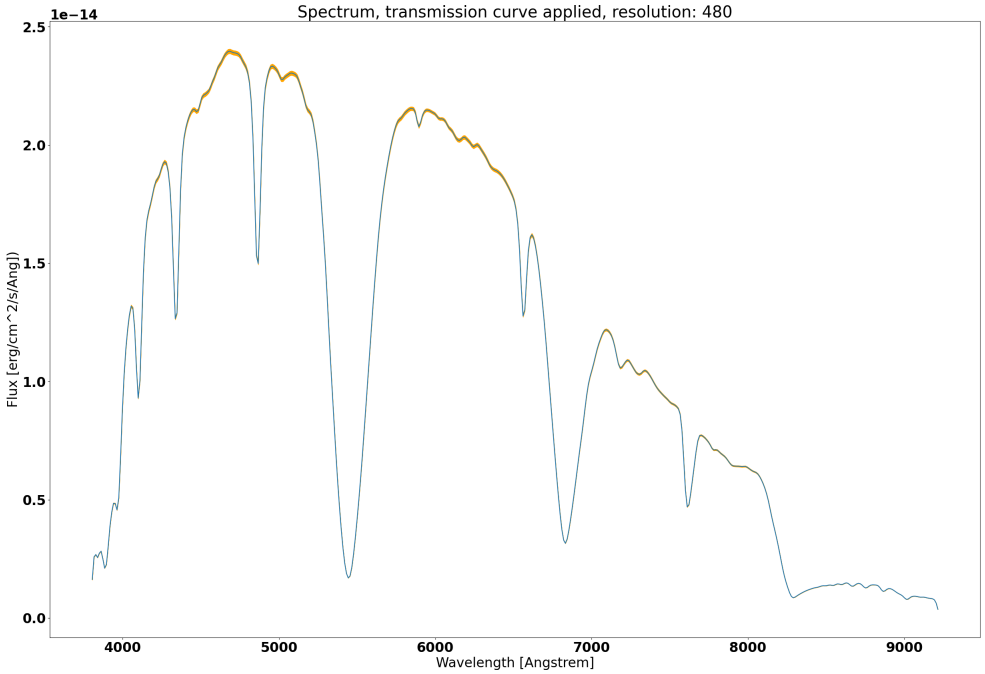

Figure A.32: This is the same plot as Fig. A.31, with the wavelength resolution halved.

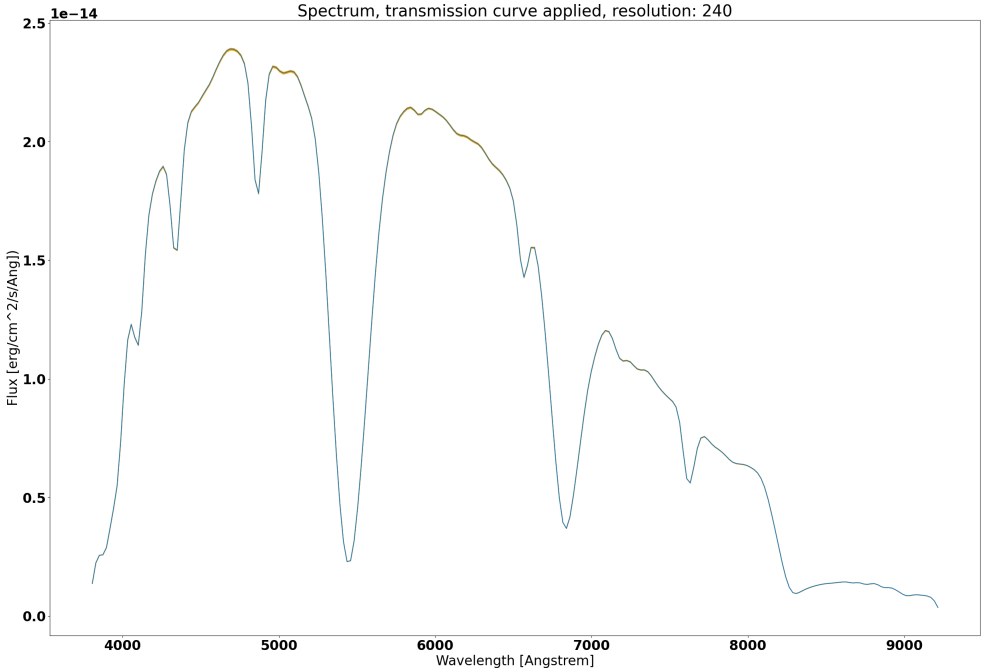

Figure A.33: This is the same plot as Fig. A.32, with the wavelength resolution halved.

A.7 Construction of down-sampled images

In this section, we describe how the down-sampled images are constructed. Cubic interpolation is used to construct the lower resolutions. We construct 4 down-sampled resolutions.

```
img_zoom_cnt = 4

fits_header, image_cube = photometry.get_multiple_resolution_image(test_image,
    img_zoom_cnt)
```

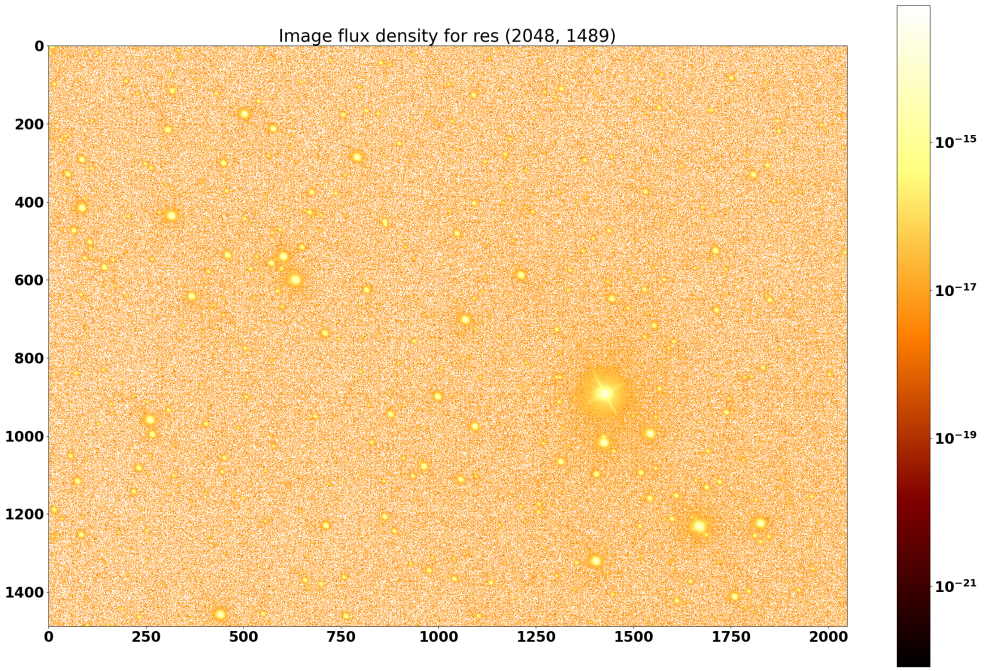

Figure A.34: This image shows the mean values of light intensities for every pixel in flux densities.

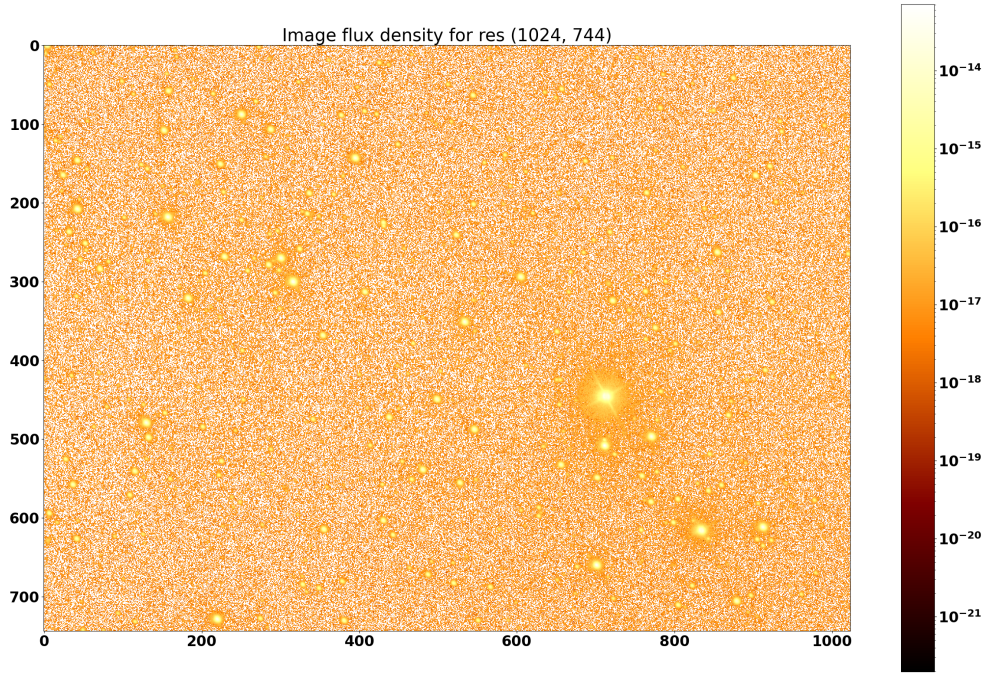

Figure A.35: This is the same plot as Fig. A.34, with the resolution resolution halved in each spatial dimension.

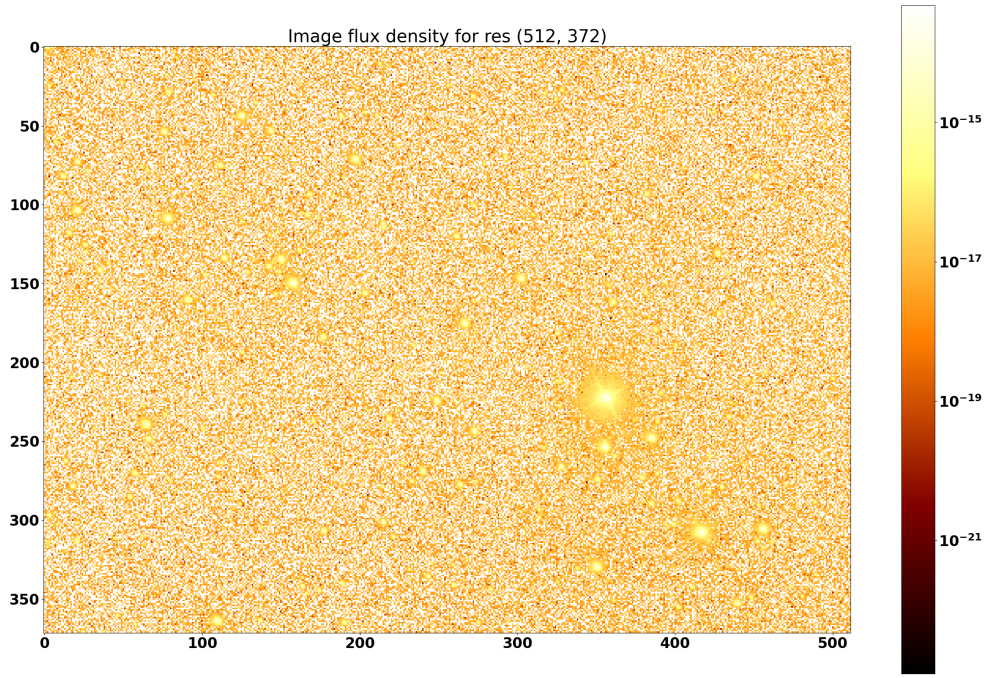

Figure A.36: This is the same plot as Fig. A.35, with the resolution resolution halved in each spatial dimension.

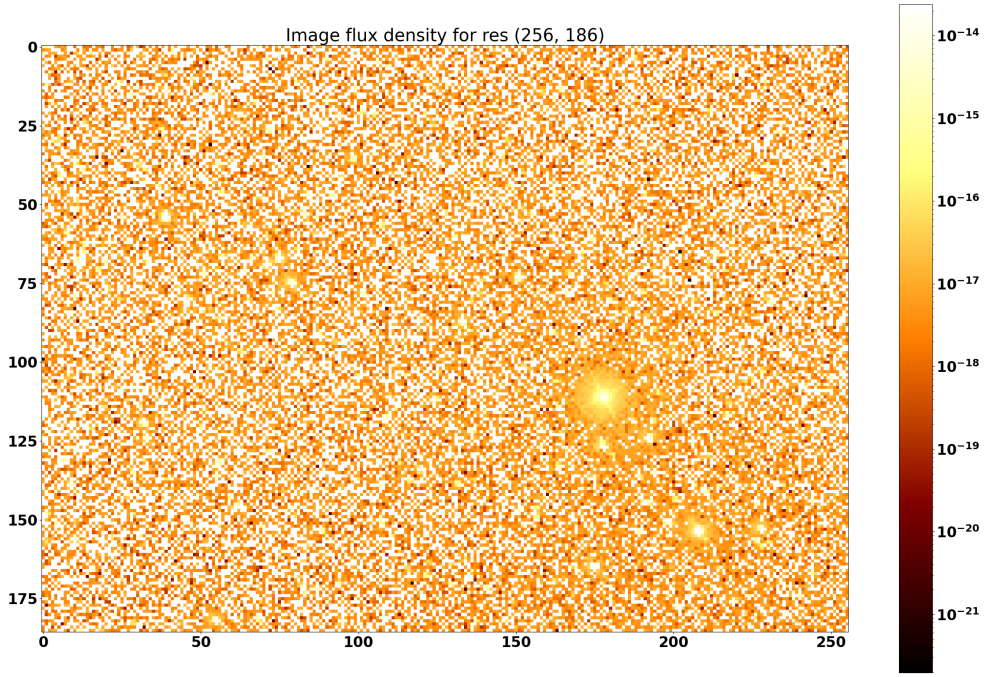

Figure A.37: This is the same plot as Fig. A.36, with the resolution resolution halved in each spatial dimension.

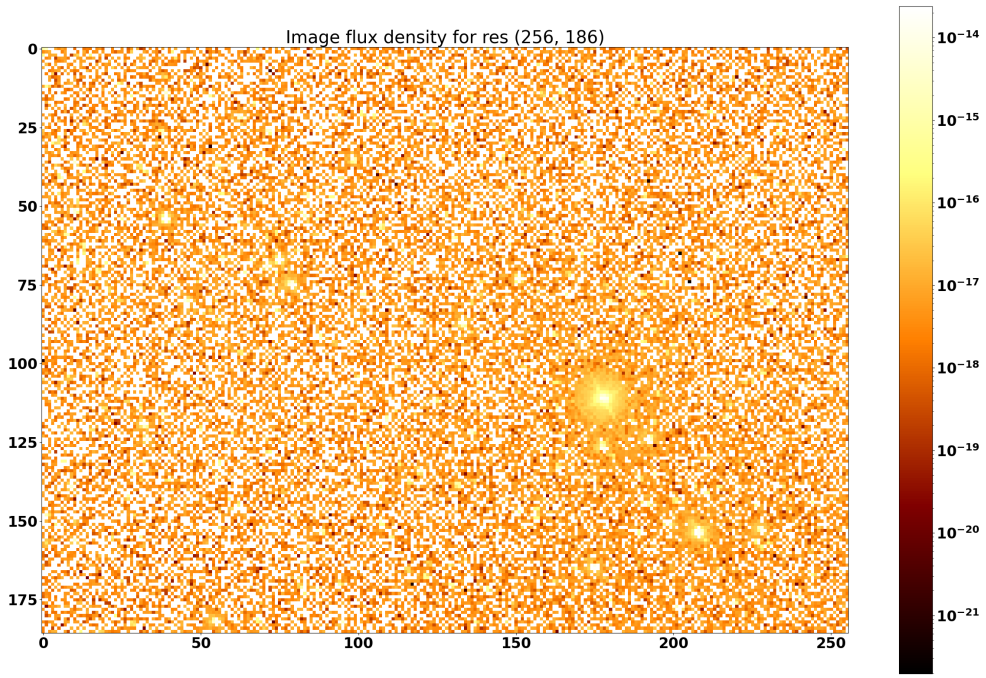

Figure A.38: This is the same plot as Fig. A.37, with the resolution resolution halved in each spatial dimension.

A.8 Construction of database in HiSS-Cube File

This section describes how HiSS-Cube is used from the command line interface. This involves both *Database construction* and *Global DB query* steps. The *Global DB query* in this example builds a 3D data cube for Machine learning purposes.

Steps involved:

1. Import of Image headers into the metadata cache.
2. Import Spectra headers into the metadata cache.
3. Transformation of Image metadata and allocation of image datasets.
4. Transformation of Spectra metadata and allocation of the spectra datasets.
5. Preprocessing and writing of Image data.
6. Preprocessing and writing of Spectrum data.
7. Linking of spectra to images.
8. Database query (construction of ML cube) from the combined images and spectra.
9. Shrinking of the ML cube to remove blank areas.

```
import warnings
from astropy.utils.exceptions import AstropyWarning

h5_output_path = "../data/processed/galaxy_small.h5"
input_folder = "../data/raw/galaxy_small"

warnings.simplefilter('ignore', category=AstropyWarning)
command = "python %s/hisscube.py %s %s create" % (module_path, input_folder,
    h5_output_path)
print("Running command: %s" % command)
!mpiexec -n 8 $command
```

```
Running command: python /home/caucau/SDSSCube/hisscube.py
    ../data/raw/galaxy_small ../data/processed/galaxy_small.h5 create

Image headers: 185it [00:00, 728.52it/s]
Spectra headers: 11it [00:00, 287.49it/s]
Image metadata: 100% 185/185 [00:00<00:00, 362.41it/s]
Spectra metadata: 100% 11/11 [00:00<00:00, 507.25it/s]
Image data: 100% 185/185 [00:34<00:00, 5.30it/s]
Spectrum data: 100% 11/11 [00:01<00:00, 7.59it/s]
Linking spectra: 100% 11/11 [00:16<00:00, 1.54s/it]
ML cube spectra: 100% 2/2 [00:00<00:00, 2.69it/s]
Shrinking ML cube: 100% 1/1 [00:00<00:00, 49.74it/s]
```

Appendix B

Online data

The prototype is available at GitHub `https://github.com/nadvornikjiri/HiSS-Cube` and the dataset used is available at Zenodo `https://doi.org/10.5281/zenodo.4273993`. The data on Zenodo contains SDSS test data, the HDF5 file tested in Chapter 8, and the exported FITS file that can be visualized in TOPCAT.