UNIVERSITY OF CAPE TOWN

DOCTORAL THESIS

# A scalable database model of RFI data for the MeerKAT/SKA Radio Telescope

*Author:*
Gerald Nathan BALEKAKI

*Supervisor(s):*
Assoc. Prof. Michelle KUTTEL
Assoc. Prof. Sarah BLYTH
Dr Anja SCHROEDER


*A thesis submitted in fulfillment of the requirements*
*for the Degree of Doctor of Philosophy*

*in the*

Department of Computer Science
Faculty of Science

May 26, 2024

# Declaration of Authorship

I, Gerald Nathan BALEKAKI, declare that this thesis, titled "A scalable database model of RFI data for the MeerKAT/SKA Radio Telescope" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

*"The illiterate of the 21st century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn."*

– Alvin Toffler

*Dedicated to my beloved family, specifically my parents: Mrs. Jane Amooti Mukasa and Mr. Peter Mukasa.*

# *Acknowledgements*

My sincere thanks are offered to the Computer Science and Astronomy departments at the University of Cape Town (UCT) and to my supervisors, Associate Professor Michelle Kuttel, Associate Professor Sarah Blyth, and Dr Anja Schroeder, for their motivation and support, without which it would not have been possible to complete this work. I am also thankful to Professor Sonia Berman for her additional support in big data and large databases.

This PhD was financially supported by the SARAO NRF Scholarship, and for the last part, by the Hasso Plattner Institute for Digital Engineering through the HPI Research School at UCT. I am immensely grateful for their full financial support for all the years.

I thank the lecturers at the HPI Research School at UCT, particularly Associate Professor Hussein Suleman, Associate Professor Melissa Densmore and Associate Professor Maria Keet, for their constructive feedback during research meetings, workshops and presentations.

I also wish to acknowledge SARAO and SAAO for the seamless collaboration. A special thanks go to the RFI Working Group, especially to Carel van der Merwe, Braam Otto and Gideon Wiid, for the technical knowledge on telescopes and RFI shared during the monthly meetings.

To my lab cohorts: Joan Byamugisha, Haffeni Mthoko, Jecton Anyango, Christine Wanjiru, Tezira Wanyana, Zola Mahlaza and Sarah Dsane-Nsor, I am deeply grateful for lessons and indescribable moments shared on this journey with you. Walking the journey filled with toil, sweat and tears of joy is one of the best things to ever happen to me.

I am also indebted to my beloved family who constantly demonstrated patience and unwavering support. To my dear parents: thank you for planting a seed of patience, wisdom and dedication with due diligence and integrity in me.

Above all, I thank God for the blessings of good health, mental acuity and energy, especially during the peak times of the Covid-19 pandemic. I could continue to work conscientiously on the research project, despite the circumstances.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **RFI** | **R**adio **F**requency **I**nterference |
| **KAT** | **K**aroo **A**rray **T**elescope |
| **SKA** | **S**quare **K**ilometre **A**rray |
| **SARAO** | **S**outh **A**frican **R**adio **A**stronomy **O**bservatory |
| **NRAO** | **N**ational **R**adio **A**stronomy **O**bservatory |
| **SAAO** | **S**outh **A**frican **A**stronomical **O**bservatory |
| **LSST** | **L**arge-aperture **S**ynoptic **S**urvey **T**elescope |
| **LOFAR** | **Lo**w-**F**requency **Ar**ray |
| **SARAS** | **S**outh **A**frican **R**adio **A**stronomy **S**ervice |
| **AGA** | **A**stronomy **G**eographic **A**dvantage Act |
| **ICASA** | **I**ndependent **C**ommunications **A**uthority of **S**outh **A**frica |
| **ITU** | **I**nternational **T**elecommunication **U**nion |
| **EM** | Electro**m**agnetic |
| **RF** | **R**adio **F**requency |
| **dB** | **D**eci**b**el |
| **SSR** | **S**econdary **S**urveillance **R**adar |
| **DME** | **D**istance **M**easuring **E**quipment |
| **ANC** | **A**daptive **N**oise **C**ancellation |
| **RTA** | **R**eal-Time **A**nalyzer |
| **HDF** | **H**ierarchical **D**ata **F**ormat |
| **HDFS** | **H**adoop **D**istributed **F**ile **S**ystem |
| **JSON** | **J**ava**S**cript **O**bject **N**otation |
| **DBMS** | **D**ata**b**ase **M**anagement **S**ystems |
| **RDBMS** | **R**elational **D**ata**b**ase **M**anagement **S**ystems |
| **ACID** | **A**tomicity, **C**onsistency, **I**solation, and **D**urability |
| **BASE** | **B**asically **A**vailable, **S**oft state, **E**ventual consistency |
| **YCSB** | **Y**ahoo **C**loud **S**erving **B**enchmark |
| **ERD** | **E**ntity-**R**elationship **D**iagram |
| **UML** | **U**nified **M**odeling **L**anguage |

# *Abstract*

In radio astronomy, radio frequency interference (RFI) refers to any signal captured by a radio telescope that did not originate from the observed target in the sky. RFI from terrestrial and other sources is a recognized problem that contaminates the desired signal and must be tracked and ultimately removed. RFI corrupts observed data and may even damage radio telescope equipment. Astronomers, therefore, seek to store data on RFI to mitigate or prevent future interference events. At the MeerKAT radio telescope (a precursor to the Square Kilometre Array, and one of the largest and most sensitive radio telescopes in the world to date), RFI is captured in different formats using a variety of devices including telescopes, sensors and scanners; however, the combination of data from these multiple sources does not only yield storage problems but also data integration challenges.

In this work, we present two designs for the scalable database model. In the first design, RFI data is stored in multiple databases (PSQL, SciDB, and Accumulo). Our findings indicate that PSQL outperforms both SciDB and Accumulo. Consequently, in the second design (alternative), all RFI data is stored exclusively in PSQL. However, we observed that the performance of the alternative model is impacted by the transformation of SciDB (array data) and Accumulo (key-value data) into PSQL (relational data). Our results recommend storing RFI data in its appropriate database rather than transforming it into another format, as this approach boosts the model's performance.

Our model demonstrates a response time of less than 12 seconds for 1 MB RFI data (bulk request), with latency below 0.14 seconds—well within the acceptable maximum latency of 1 second in scalable databases. We found that the native database API is slightly faster (5%) than a third-party API, with no significant impact on the model's performance. In addition, this work indicates the direction for improvement in join queries involving disparate databases, which remains a limitation in heterogeneous environments. Our model facilitates fast queries across various databases, underscoring the importance of storing each data type in the appropriate database system. Lastly, our RFI database model offers good performance and scales effectively with increasing data volumes, multiple users, and varying workloads, making it suitable for the MeerKAT and SKA radio telescopes.

# Chapter 1

# Introduction

The Square Kilometre Array (SKA) telescope project aims to construct one of the largest and most sensitive telescope in the world, with the first phase (about 10% of the planned full SKA) expected to complete in 2023 [1]. The SKA precursor, MeerKAT [2], is currently in operation in South Africa and comprises approximately 1% of the final SKA, but is already one of the most sensitive radio telescopes in the world [3]. The full SKA is expected to produce data volumes equal to the current global total internet traffic [4].

RFI is an important issue for radio telescopes, such as the MeerKAT/SKA. Radio frequency interference refers to any signal captured by a radio telescope that did not originate from the observed target in the sky. RFI sources range from human-generated ones to natural objects such as the sun. RFI signals are typically stronger than the weak celestial signals of interest and therefore have a dramatic deleterious effect on observation data [5]. With the expansion of technology, RFI contamination of the radio signal is constantly growing with time. Additionally, the radio spectrum allocated to radio astronomy is limited, the greater part of the spectrum is reserved for commercial purposes [6]. As a result, radio astronomical observations are susceptible to human-made RFI.

Figure 1.1 illustrates potential sources of RFI likely to interfere with radio observations, such as television (TV), FM radio, digital audio broadcasts (DAB), satellite communication, cellular networks (GSM, UMTS), wireless computer networks (WLAN). While some sources of RFI originate from legally allocated communication services licensed by regulatory authorities ICASA in South Africa) [7], the majority remain unidentified.

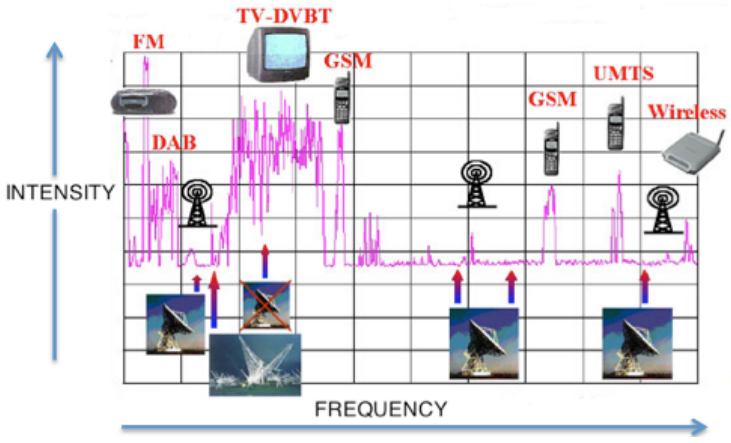



FIGURE 1.1: Potential RFI sources in radio astronomy [8].

There is no complete solution to eliminating RFI, but several approaches have

been introduced to reduce the damage on celestial signals [9, 10, 11]. These approaches range from regulatory to technical methods. The first approach in RFI mitigation is prevention: this is provided for explicitly in the Astronomy Geographical Act (AGA) of South Africa, which protects areas suited to radio astronomy [12]. The second approach involves the detection and monitoring of RFI signals. Monitoring RFI at MeerKAT is implemented using two devices: a fixed monitoring system that provides continuous monitoring, and a mobile monitoring system that resides on a vehicle that can be deployed anywhere within or beyond the telescope array [13].

Radio astronomers collect and store RFI data to explore and understand the nature of the RFI sources. The RFI data are dynamic and diverse, including measurement sets, arrays, tables, text, spectral images, and JavaScript Object Notation (JSON) [14]. Such complex data sets vary widely in quality, coverage, accuracy, and period, making data storage a concern: diverse data impacts data interpretation and is cumbersome [15]. The advantage of collecting RFI in different formats is that it provides a subtle and detailed picture of the nature and source of the interference. In addition, RFI data collected in isolation (from MeerKAT and other international radio telescopes) hinders data integration [16]. This problem is likely to be exacerbated in the future as a consequence of high data volumes generated at high rates during large survey observations at the SKA in the future.

Traditional flat-file formats, such as CSV, are commonly used by many scientists for their data storage. However, flat files lack the structure necessary for indexing multiple files [17]. Structured file systems, such as Hierarchical Data Format Version 5 (HDF5) [18] and Network Common Data Form (NetCDF) [19], have been introduced to deal with large and diverse scientific datasets but are still built on the basic principle of flat files [17, 20] and do not fully support indexing, caching, structure-like queries, reliability and scalability [21], essential for many science applications that require real-time support, cross-platform querying, detail analytics, and data visualization [22].

A database management solution is preferable, but the most serious challenge in using databases to store scientific data lies in integrating several data elements. Previously, Fridman et al [10]. proposed the creation of an RFI database at each radio telescope for further reference of the RF environment to ensure that the environment around the telescope is free from RFI signals. However, the challenge faced by the traditional relational database models (RDBMS) is the inability to scale and integrate data in multiple formats without compromising the essential READ and WRITE database operations. In particular, using RDBMS for storing RFI would mean that only well-structured data that fits appropriately into tables would be stored.

Previously, RDBMS would handle only structured data. Today, RDBMS support semi-structured data like JSON and XML. However, unstructured data, such as images, scanned documents, sensor data, audio, and video, are still not supported. Therefore, unstructured data would have to be discarded, despite all the valuable information they might contain. In general, a large portion of the science data comprises multi-dimensional structures that are inadequate for the traditional 2D relational models.

Previous studies on scientific databases have recommended PostgreSQL (PSQL) [23] and SciDB [22] as solutions for astronomy. PSQL has been used in astronomy as it supports complex data objects and astronomy-specific functions [24]. SciDB has been used in EarthDB to store moderate resolution imaging spectroradiometer data on earth dynamics and supports fast searches, bulk loading, and ad-hoc analysis [25]. Stonebraker et al. [21] have demonstrated SciDB on imagery from the

Large-aperture Synoptic Survey Telescope (LSST) [26] using the array data model to find space objects, such as galaxies. However, this supports a single modality of data that disregards several data models found in scientific data.

We focus on radio astronomical data, specifically on RFI. Our primary focus is on efficiently storing RFI data rather than identifying it. We argue that efficient RFI storage is a fundamental prerequisite for successful RFI identification. In our approach, we emphasize the use of database systems over traditional file systems. While file systems can handle large volumes of data, they lack the comprehensive database functionalities required for RFI data management. Therefore, our research centers on new and scalable database models, particularly Scalable SQL (PSQL), NewSQL (SciDB), and NoSQL (Accumulo). These models offer scalability without compromising performance, making them well-suited for managing the substantial volume of RFI data.

It's worth noting that our focus is on MeerKAT data collected before and during the construction of the telescope, which consists of 64 antennas. We do not consider data from the ongoing SKA international telescope project, which involves an additional 133 antennas. At MeerKAT, RFI interference from terrestrial and other sources has been a persistent issue that contaminates the astronomical signals of interest. RFI is recorded by various devices, including telescopes, sensors, and scanners. However, integrating data from these diverse sources into a unified RFI database model presents a significant challenge. Addressing this challenge holds the potential to greatly benefit radio astronomers by improving RFI coordination and analysis, consequently enhancing the quality of radio astronomical observations.

To address the primary research question, we initiated a requirements survey targeting potential users of the RFI database, including radio astronomers, engineers, and computer scientists at the MeerKAT radio telescope in South Africa. This diverse group of scientists was selected for their ability to provide valuable insights into the specific needs of the RFI database. Through this survey, we collected, analyzed, and interpreted qualitative data to gain insight into the RFI storage challenges at the MeerKAT radio telescope.

We designed a database based on the foundational principles of conceptual, logical, and physical database models. The conceptual database model outlines the core entities necessary in the RFI database environment and their relationships. The logical database model provides detailed specifications of these entities, including attributes, primary keys (unique entity identifiers), and foreign keys (establishing relationships between entities). Finally, the physical database model translates the logical model into an actual DBMS implementation, specifying how data is stored and accessed. Therefore, this comprehensive approach enables us to assess the scalability and performance of the database models in handling RFI data effectively at the MeerKAT radio telescope.

Subsequently, we evaluate these database models in data-intensive settings, including scenarios with increasing data volumes, multiple users, varying workloads, bulk uploads, and API environments. In an environment with increasing data volumes, data records double over time, creating an exponential growth pattern. The multi-user environment simulates situations with a growing number of concurrent users who can access the database simultaneously. Varying workloads encompass a combination of database operations, such as Inserts, Reads, and Updates. The bulk upload environment tests the ingestion of large data files into the database, evaluating its capability to handle significant data loads. Finally, API environments involve two methods of accessing the database: a native API, inherent

to the database model, and a third-party API, which is web-based and external to the model.

Finally, the thesis is structured as follows. In Chapter 1, we explore modern radio astronomy and define RFI and how it can be mitigated. We then focus on the MeerKAT radio telescope in South Africa and discuss the contextual inquiry of the RFI storage problem in Chapter 2. In Chapter 3, we present existing database models suitable for this project. Further, we highlight the challenges each model encounters with the current needs of science data. In Chapter 4, we describe the database designs and the research approach. Here the thesis focuses on three design guidelines involving the conceptual, logical and physical models while designing the RFI database. In Chapter 5, we describe the experimental design and the test environments. Our results and related discussions are presented for each experiment in Chapter 6. Finally, in Chapter 7, conclusions and possible directions for future work are discussed.

# Chapter 2

# Background

In this chapter, we discuss the context of this work in light of current literature. We start by describing modern radio astronomy and the next-generation radio telescopes, such as the MeerKAT/SKA radio telescope in South Africa. We describe radio frequency interference (RFI) in radio astronomy data, focusing on the existing storage challenges. Lastly, we describe RFI mitigation approaches for radio astronomy.

## 2.1 Modern Radio Astronomy

In radio astronomy, radio observations offer numerous benefits for understanding the Universe. Astronomers utilize radio waves emitted by celestial objects to gain invaluable insights into fundamental processes and the formation of the Cosmos. Radio observations enable scientists to study distant objects, including stars, galaxies, pulsars, quasars, and interstellar objects [27]. Moreover, radio waves can penetrate cosmic dust, interstellar gas, and Earth's atmosphere, unveiling insights into various astrophysical phenomena such as black holes, star formation, and galaxy evolution that would remain unknown in visible light [28]. Additionally, the extended radio observation window is not hindered by sunlight interference, enabling continuous data collection over a wide range of frequencies and facilitating the exploration of transients [29]. Lastly, radio observations have enabled scientists to conduct comprehensive multi-wavelength studies. By combining radio observations with observations from other wavelengths, researchers have made significant discoveries and unraveled many mysteries of the Universe. For example, radio observations can complement optical or X-ray observations of galaxies, allowing the detection of unique objects like pulsars or fast radio bursts (FRBs), as well as enabling cosmological observations of neutral hydrogen using the 21 cm signal [30, 31].

For over five decades, astronomers have been able to measure radio waves using telescopes. A radio telescope is an instrument that consists of a receiver system and an antenna that focuses the waves on the focal point of the receiver (see Figure 2.1).

FIGURE 2.1: A radio telescope at the South African Radio Astronomy Observatory, Northern Cape [32].

Today, modern astronomers use highly sensitive radio telescopes to detect and study weak celestial radio emissions. The electromagnetic (EM) spectrum ranges from short-wavelength radiation (gamma rays and x-rays) to long-wavelength (radio waves). However, a large portion of the EM spectrum is absorbed by the Earth's atmosphere, blocking the EM radiation from reaching the Earth's surface (see Figure 2.2).

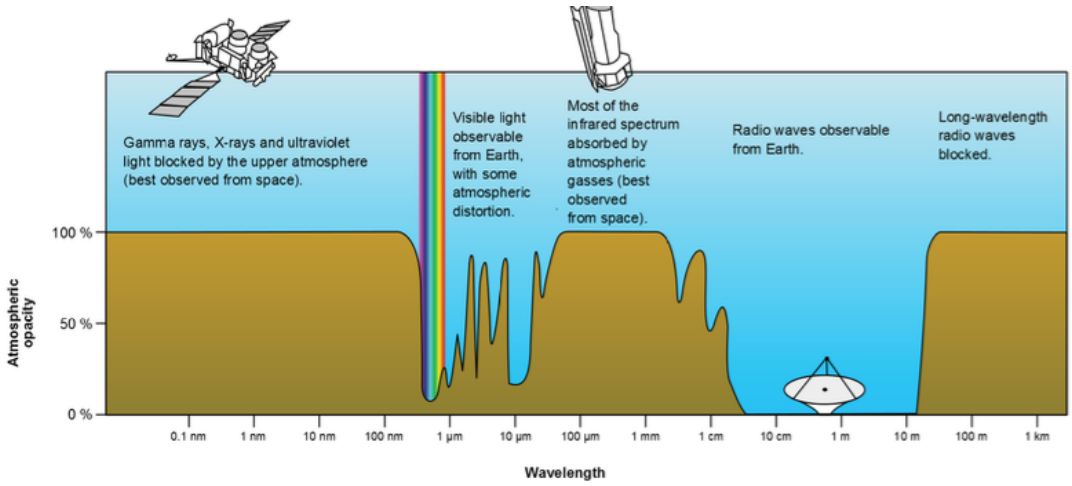



FIGURE 2.2: Earth's atmospheric transmittance at different wavelengths [33].

The Earth's atmosphere is transparent to visible light and radio waves with frequencies (wavelengths) of 30 MHz to 300 GHz (or 10 m to 1 cm), which is, therefore, suitable for ground-based telescope observations.

In 1933, Karl G. Jansky [34] detected radio frequency interference (RFI) from unknown origins. RFI refers to the undesired radio signals captured by a radio

telescope while observing celestial objects. These signals corrupt the observed data. Jansky further found that the strongest RFI emission was coming from the center of the Milky Way galaxy. His study was, however, limited to low frequencies (approx. 20.5 MHz), but was later (in 1940) verified by Grote Reber [35]. Reber created the first radio map of the Galaxy observed at higher frequencies (approx. 160 MHz) than Jansky.

Modern telescopes operate over a wide range of frequencies (30 MHz to 300 GHz). For example, the MeerKAT radio telescope with its low surface brightness sensitivity can image sources not seen in previous observations by other telescopes (MeerKAT International GHz Tiered Extragalactic Exploration – MIGHTEE survey) [36].

## 2.2 Radio Telescopes

A single-dish telescope is a radio telescope that consists of one radio antenna. The world's largest single-dish radio telescope is the Five-hundred-meter Aperture Spherical Telescope (FAST) [37]. It consists of a fixed 500-meter diameter dish located in a natural depression landscape in the Guizhou Province in southern China [38]. The largest fully steerable (capability to observe in any direction) single telescope in the world is known as the Robert C. Byrd Green Bank Telescope (GBT) [39, 40]. It measures 110 meters and is located at Green Bank, West Virginia, USA. The size of a radio telescope is limited by mechanical considerations and costs.

To improve the resolution of the telescope while maintaining sensitivity, astronomers require larger apertures. This can be achieved through the use of an arraying technique known as an interferometer [41]. The separation between two antennas in the interferometric array is referred to as the baseline [42, 43]. By combining signals from multiple antennas, an interferometer creates an effective aperture equivalent to the distance between the antennas, which is impractical to build with a single antenna telescope [44, 45]. The antennas measure the phase and amplitude of the combined signals to extract information about the location, size, and spectral characteristics of celestial objects. Interferometers provide high-resolution and sensitivity, enabling scientists to conduct detailed studies across a wide range of astrophysical phenomena, from distant galaxies to pulsars and cosmic microwave background [46]. Further advancements in interferometric techniques, coupled with the construction of next-generation array telescopes like the MeerKAT/SKA radio telescope, hold great promise for cutting-edge discoveries and a deeper understanding of the Universe.

Some of the famous interferometers in radio astronomy include the Very Large Array (VLA) [47], Westerbork [48] and the Australian Telescope Compact Array (ATCA) [49]. The Australian SKA precursor instruments include the mid-frequency Australian SKA Pathfinder (ASKAP) and the low-frequency Murchison Wide-field Array (MWA) located at the Murchison radio-astronomy observatory in Western Australia [50]. The MeerKAT radio telescope, located in South Africa, along with the Hydrogen Epoch of Reionization Array (HERA), serves as a precursor to the Square Kilometer Array (SKA) [51, 52]. The MeerKAT has been fully constructed and will eventually become an integral part of the international SKA telescope, which is set to be one of the largest and most sensitive radio telescope worldwide [53]. HERA is dedicated to studying the large-scale structure during the pivotal period when the Universe transitioned from a neutral to an ionized state known as the Epoch of Reionization (EoR). It consists of an array of low-frequency antennas

designed to detect the faint radio signals emitted by neutral hydrogen during the EoR [52]. With its large collecting area, HERA enables the detection of weak signals, while its wide field of view facilitates efficient mapping of extensive sky regions. Precise calibration techniques are employed to mitigate instrumental effects and errors. HERA has provided significant scientific insights into 21-cm cosmology, power spectrum measurement of neutral hydrogen, and cross-correlation studies, contributing to our understanding of cosmic dawn and the EoR [54].

Although interferometers are fully steerable and highly sensitive, they present several infrastructure and computing challenges. For instance, the cost of equipment for building an interferometer is exceptionally high [55]. Further, the computational load for SKA is predicted to be very large (150 Petaflops [1]), which is costly to process considering computer software, memory and computational time [57].

## 2.3 Primary Beam and Field of View

The response of a radio telescope to a point source is not uniform across the entire sky. The beam of an ideal paraboloid is just the Airy pattern with a central Airy disk (main lobe) comprising 83% of the radiation surrounded by fainter concentric rings (side lobes) [58]. Airy patterns in the main lobe of antennas are diffractive patterns characterized by the wave nature of light and the limited size of the antenna aperture, manifesting as bright and dark rings surrounding a central bright spot. The beam of the interferometers is more complex. The instrument resolution is usually measured by the width of the main lobe at half power (Half Power Beam Width, HPBW). Further, the resolution of the instrument of diameter $D$ depends on the ratio $\lambda/D$, where $\lambda$ is the wavelength. The primary beam from the main lobe is Gaussian, (Full Width Half Maximum, FWHM) approx. $1.2\lambda/D$, and the limited field of view is approx. $\lambda/D$.

Figure 2.3 illustrates a longitudinal section of the beam showing received power in polar coordinates as a function of the angle of arrival of the signal, $\theta$. In the Figure 2.4, the main lobe is identified from the intensity of the central Airy disk and sidelobes from the intensity of the Airy rings. In general, the first sidelobe is a few percent of the peak, but likely to increase as a result of aperture obstructions or other surface inaccuracies. Consequently, this degradation in image quality arises from the trade-off between reducing sidelobes to a minimum while sacrificing the effective area. Ideally, an interferometer's beam exhibits high sidelobes (approximately 15%), although certain numerical modifications can be employed during the imaging process to mitigate this effect [58].

[1] A petaflop is one thousand million million floating-point operations per second of raw processing power [56].

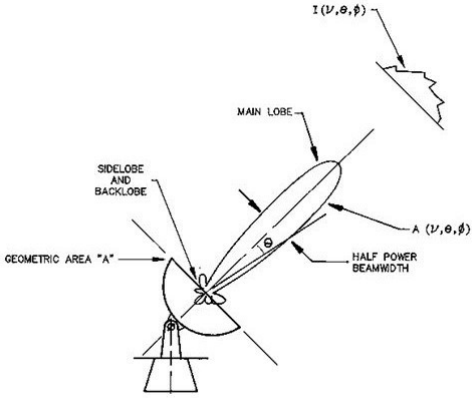



FIGURE 2.3: A representation of the primary beam [59]

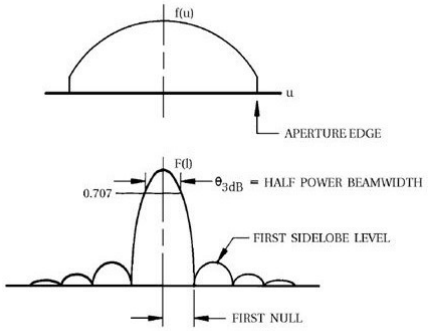



FIGURE 2.4: Field of view, taken from [59].

In interferometry, the visibility function is used to measure the visibility in relation to the sky. Let's consider the distance between two antennas, denoted as $p$ and $q$, also referred to as the baseline. The coordinates $u$, $v$, and $w$ represent the vector components of the baseline in units of wavelength. The $u$ and $v$ coordinates are oriented towards the east and north directions, respectively, while the $w$ coordinate points towards the phase center of the observed field [60]. The Van Cittert-Zernike therom [61, 41] presents the nominal observed visibility as

$$V^{nom}(v,u) = \int_l \int_m \frac{I_o(l,m)}{\sqrt{1-l^2-m^2}} e^{-2\pi i \phi(u,v,w)} \{d\} l \{d\} m. \tag{2.1}$$

Given $I_o(l,m)$ as sky distribution, l, m as direction cosines, $\phi(u,v,w) = ul + vm + w(n-1)$, and $n = \sqrt{1-l^2-m^2}$. The correlator introduces a compensating delay to ensure $\phi = 0$ at the center of the field. When the fringe stopping is in effect, the term $n-1$ is introduced instead of the term $n$. Consider a pair of antennas $p$ and $q$ forming a baseline $u_{pq} = (u_{pq}, v_{pq}, w_{pq})$, the equation for the visibility data in relation to the primary beam patterns $\varepsilon_p(l,m)$ and $\varepsilon_q(l,m)$ , which define the directional sensitivity of each of the two antennas, is given by

$$V_{pq}(v,u) = \int_l \int_m \frac{\varepsilon_p I_o \varepsilon_q^H}{\sqrt{1-l^2-m^2}} e^{-2\pi i \phi(u,v,w)} \{d\} l \{d\} m. \tag{2.2}$$

Alternatively, the effect of the primary beam can be expressed as the convolution of its Fourier transform and the aperture illumination function, $A_p(u, v)$ [60]. (See equation 2.3).

$$V_{pq} = A_p \circ V_{pq}^{nom} \circ A_q^H. \tag{2.3}$$

## 2.4 RFI in Radio Astronomy

A limiting factor for ground-based telescopes is human-generated RFI, which is often stronger (many billions of times) than the weak celestial signals of interest, drowning them out and corrupting observational data [9, 5]. Potential sources of RFI likely to interfere with radio observations are typically television (TV), FM radio, digital audio broadcast (DAB), satellite communication, cellular networks (GSM, UMTS), wireless computer networks (such as the WLAN) and air navigation systems, such as Distance Measuring Equipment(DME)) (Figure 1.1). Although some of the RFI sources are unidentified, one source of RFI includes legally allocated communication services licensed by regulatory authorities, such as the ICASA in South Africa [7].

### 2.4.1 Effects of RFI

Radio telescopes suffer from RFI in the same way that optical telescopes suffer from light pollution. Broadband RFI raises the general noise level of receivers which, consequently, degrades telescope sensitivity. Narrow-band RFI may imitate spectral lines. The human-generated RFI entering the receiver chain of the radio telescope has much higher power than the natural astronomical signals. The latter's power is generally 60 dB below the receiver noise level [62], and thus even a relatively weak man-made RFI source can completely obscure astronomical signals, which negatively affects scientific observations.

Figure 2.5 and 2.6 display observational images of the northern sky obtained from the LOFAR test station in the Netherlands [63]. The image in Figure 2.5 is an observation without RFI (transmitter), while the image in Figure 2.6 is an observation made with the presence of the transmitter, introduced at the horizon of the sky. Here, the observation has been affected by the RFI coming from the transmitter, resulting in the radio sources Cyg A and Cas A being obscured, compared to the image above in which Cyg A and Cas A are more visible.

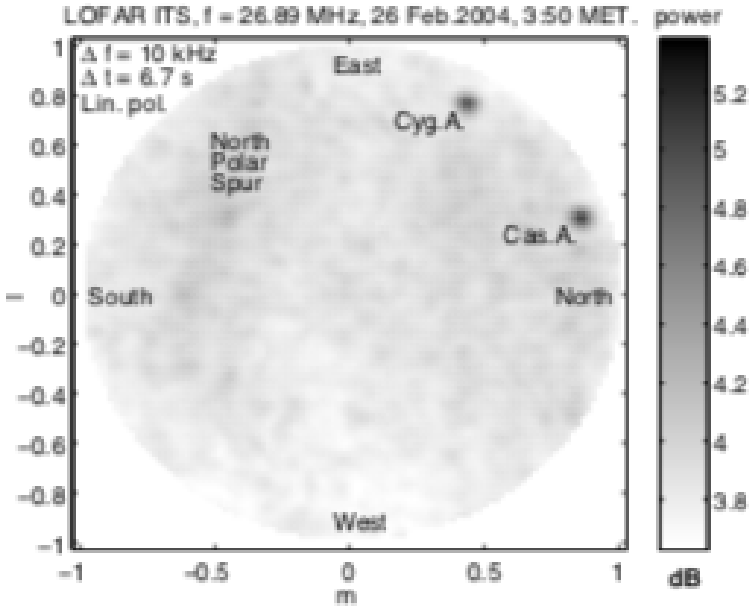



FIGURE 2.5: Effect of RFI on observations without RFI. Astronomical sources (Cyg A and Cas A) are more visible in the observation without RFI [63].

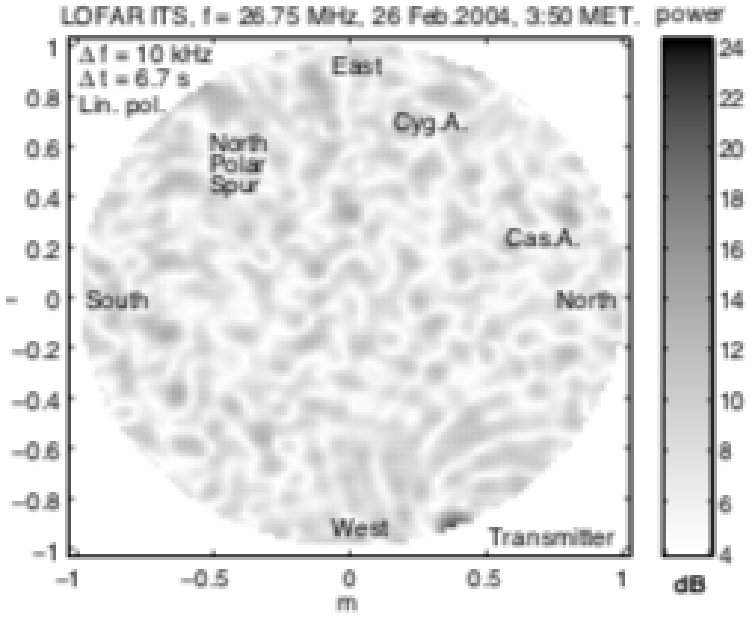



FIGURE 2.6: Effect of RFI on observations with RFI (bottom). Astronomical sources (Cyg A and Cas A) are less visible in the observation with RFI [63].

Figure 2.7 shows measurement studies conducted to guide the selection process of SKA candidate sites with low levels of interference [64]. The top panel displays signal interference as the interference-to-background-noise ratio, which is utilized to estimate the levels of interference (RFI) from 100 MHz to 2000 MHz. The interference-to-background-noise ratio (I/N) represents the ratio of the signal strength of RFI (I) to the background noise (N) [65]. There are high levels of interference (approximately 50 dB) from 100 to 600 MHz, 1000 to 1100 MHz, and

1500 to 1900 MHz frequency ranges. The rest of the channels have relatively low interference (approximately 0 dB).

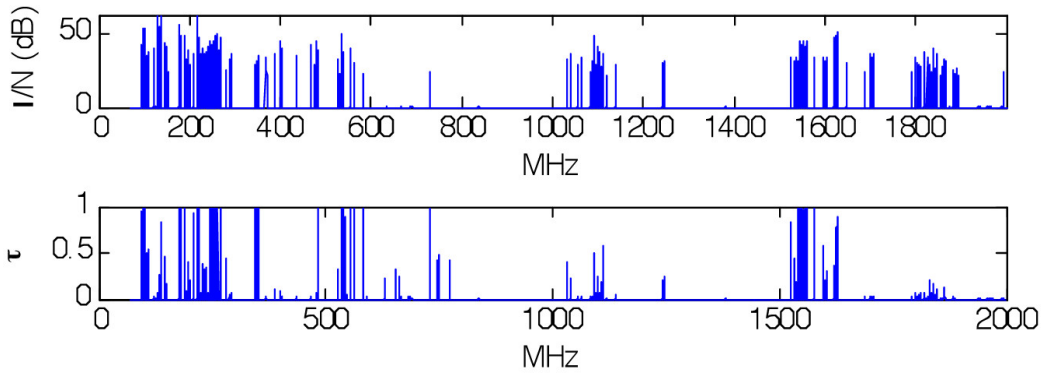



FIGURE 2.7: Effect of RFI on frequency channels (100 to 2000 MHz): RFI given in terms of signal to noise ratio (top), and channel occupancy over time (bottom) [64].

The bottom panel presents the channel occupancy of RFI from 100 MHz to 2000 MHz. Channel occupancy refers to the level of interference within a specific frequency band and is used to estimate the amount of either desired astronomical signals or unwanted interference occupying the bandwidth of a given channel [66]. The channel occupancy from 100 to 600 MHz, 1000 to 1100 MHz, and 1500 to 1900 MHz are close to one. This means that RFI emission dominates these channels. Persistent RFI can span the entire frequency channel for long periods and may block out several channels, resulting in total loss of data. Radio astronomers use occupancy statistics to determine the amount of time in which the radio telescope is available for undisturbed radio observations in a particular band. Occupancy close to zero implies that interference in those channels is less likely.

### 2.4.2 Sources of RFI

In radio astronomy, telescopes, such as the MeerKAT/SKA, operate at designated radio frequencies governed by the International Telecommunication Union (ITU). Should the same radio frequencies be used by other transmission services close to the telescopes, interference is likely. Sources of RFI are classified as either external or internal to the telescope environment. Internal RFI is interference produced by the telescope system itself. External RFI encompasses all interference generated by sources on the Earth and its satellites. We describe the two classes of RFI sources in the Sections below classified as either internal or external. These sources are further divided into two groups: intentional and unintentional RFI.

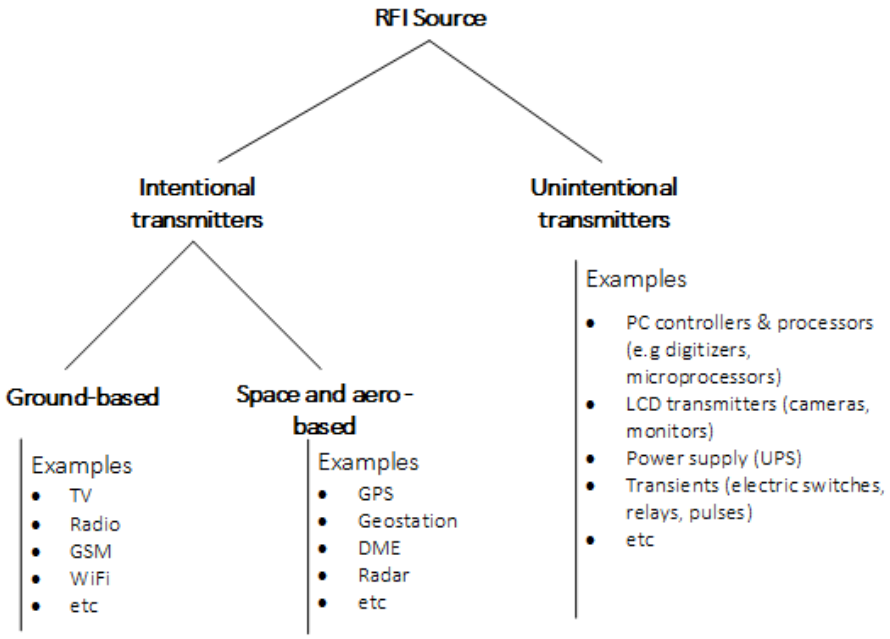



FIGURE 2.8: RFI sources divided into intentional or unintentional transmitters, taken from [67].

Figure 2.8 illustrates the two groups of RFI sources (intentional and unintentional) and also provides examples of each. Intentional transmitters are those licensed by ICASA to transmit deliberately in a designated frequency band (FM radio, TV broadcasts). Intentional transmitters fall into two groups: ground-based transmitters, which include TV, radio, GSM and WiFi, and space or aero-based transmitters which include GPS, Geostation, DME and radar. These transmitters are relatively easy to find because they have known, published frequencies and their effects on radio astronomy observations can be avoided by selecting RFI-free bands.

Unintentional transmitters are devices such as computer controllers and processors (digitizers, microprocessors), liquid crystal displays (LCD cameras & monitors), uninterrupted power supplies (UPS), and electric fences. These transmitters are difficult to detect, and their impact on the measurements is difficult to estimate [68, 69]. Such unintentional transmitters are shielded, when affecting the telescope, depending on their harmfulness. Ideally, their spectral measurements are taken to establish the harmfulness, before they are allowed to operate under strict permissions.

### 2.4.3 Internal RFI

Radio observatories are facilities that house radio telescopes and are themselves a significant source of RFI [5, 70]. These radio facilities consist of several emitting devices, such as telescope steering electronics required to operate a steerable radio telescope, fluorescent and LED lamps, computing equipment, such as personal computers and notebooks, computer screens, wiring cables, etc. Most of them are internal and required for radio astronomical studies. Therefore, they cannot be avoided, and they generate narrowband and broadband emissions. These emissions enter and interfere with the telescope detection system, especially if adequate shielding is not properly installed to attenuate the emissions [71]. Internal RFI from the radio telescope is the most persistent type of RFI and, even when very weak, can

affect the observations when not shielded. External RFI sources need to be stronger to have the same interfering effect.

### 2.4.4 External RFI

External RFI comes from the surrounding environment of the observatory. These sources are either fixed or movable and transmit at all bands, sometimes including protected bands. External RFI may occur naturally or be human-generated. For instance, naturally occurring sources of RFI include the ground, Sun, lightning, or other bright radio sources. Human-made interference may arise from broadcast services (TV, radio), voice and data communications (mobile telephones, two-way radio, wireless IT networks), navigation systems (GPS, GNSS), secondary surveillance radar (SSR), remote sensing, military systems, electric fences, car ignitions and domestic appliances (e.g., microwave ovens) [8, 5]. Ground-based RFI is often more harmful due to its high transmission power and proximity to the ground-based radio telescope. Space- and aero-based RFIs are increasing due to growing air traffic and satellite communication transmissions. Air traffic uses a significant number of frequencies for navigation and communication utilising DME and SSR radars in South Africa [67].

### 2.4.5 Characteristics of RFI

Some astronomical sources exhibit similar signal characteristics to those of RFI sources. Unintentional RFI in particular is usually transient, intermittent and broadband. Some astronomical sources, such as pulsars [72], fast radio bursts (FRBs) [73], or rotating radio transients (RRATs) [74], also emit transient [2]) signals. As a result, it is difficult to distinguish RFI from astronomical source signals [76]. Both strong and weak RFI signatures can affect specific or several adjacent channels. The RFI signature in observed data is studied in terms of its morphology, which is broadly either broadband or narrowband (Figure 2.9). A signal is broadband when distributed over many or all channels, and narrowband when confined to only a few specific channels or frequencies [77].

[2] A transient is a radio source that is not constant, either changing significantly in brightness or appearance, usually lasting for a limited period [75].

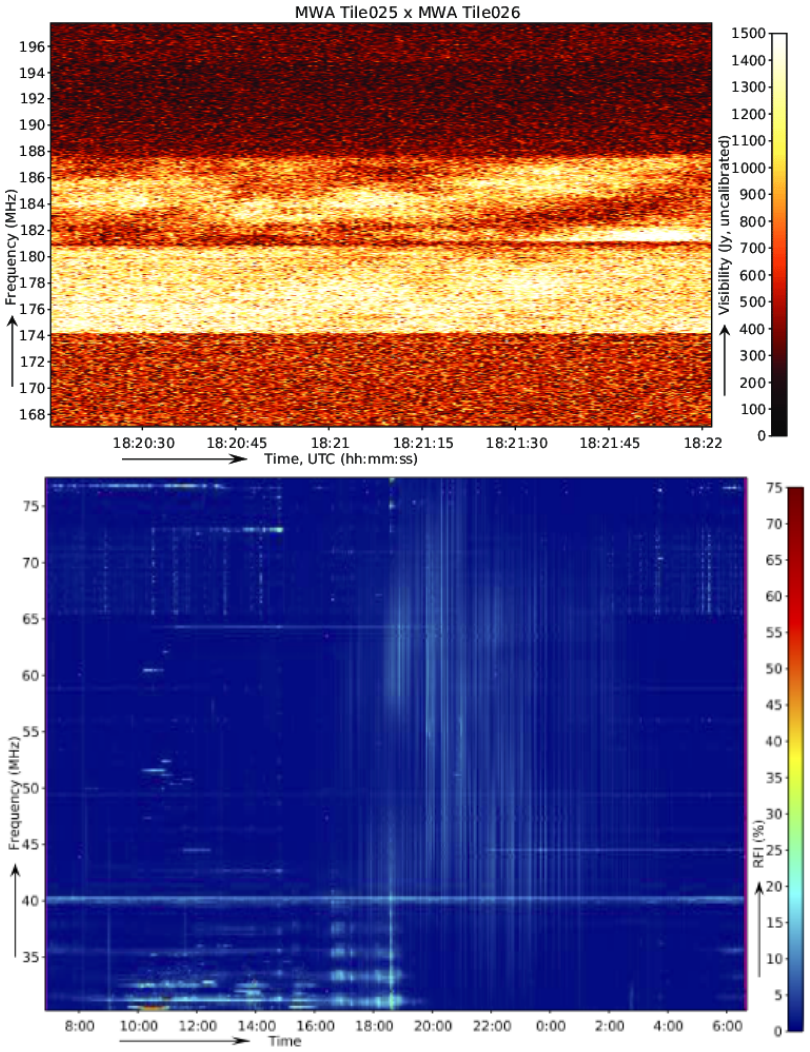



FIGURE 2.9: Top – Persistent broadband RFI from digital TV measured by MWA (Murchison Widefield Array) [78]. Bottom – The broadband RFI lasts for the entire observation time through several channels. Transient narrowband and broadband RFI by LOFAR (Low-Frequency Array) [77]. Both broadband and narrowband RFIs last for a short time before they will re-appear; this can happen in a single or many channels.

Figure 2.9 shows two axes representing time and frequency, with the plotted quantity being the magnitude of the visibility (i.e., the fundamental measurement from an interferometer), which inherently constitutes a complex quantity. The RFI contamination can be visually identified as the bright streaks in the image, indicating a large magnitude in comparison to the generally smooth and faint background signal.

Depending on the originating source and observation time during the day, the source can either be transient or persistent [77, 78]. Transient RFI sources may have similar characteristics to real astronomical transients, exacerbating the identification problem. Many RFI sources exhibit highly variable signal characteristics that follow a non-Gaussian probability distribution [10]. RFI signals vary significantly from each other in shape and hence are difficult to identify or distinguish. This is an inherent characteristic of the majority of RFI sources [10]. Table 2.1 lists some of the characteristics of common RFI detected at the SARAO [79]. SARAO is the national

center for optical and infrared astronomy in South Africa responsible for astronomy and astrophysics research.

TABLE 2.1: A summarized list of RFI sources with their characteristics

| RFI source | Characteristics |
|---|---|
| Two way radio | Narrowband, Intermittent, High-powered, Mobile, Polarised |
| GPS satellites | Narrowband, Persistent, High-powered, Stationary, Polarised |
| Electric fences | Broadband, Burst-like, power dependent on distance, Polarised |
| Ethernet cables | Broadband, Intermittent, High-powered, not Polarised |
| Lightning | Broadband, Burst-like, High-powered, not Polarised |
| Television | Narrowband, Persistent, High-powered, Stationary, Polarised |

An intermittent source transmits on and off for periods of minutes to hours. A burst-like RFI source transmits in short bursts in the order of milliseconds to seconds. A High-powered RFI is more powerful than the instrument noise, while a low-powered RFI has similar power to the instrument noise. Stationary RFI characteristic is fixed in one location. Polarised RFI has a definite direction relative to the direction of propagation of the EM waves. Each RFI characteristic affects the observed data differently, making mitigation difficult [10, 5].

### 2.4.6 Harmful RFI Levels

Radio telescopes are intrinsically exceptionally sensitive to RFI, and the presence of RFI (internal or external) can affect radio astronomical observations in several ways and be detrimental to a radio astronomical receiver. Single-antenna observations are more susceptible to RFI than interferometer (array telescope) observations that cancel out some RFI due to the separation or distance between antennas. For instance, in modern amplifiers a gallium arsenide field-effect transistor (GaAsFET) and a high-electron-mobility transistor (HEMT) can be damaged by a signal power of 0.1 and 0.01 Watts, respectively [80]. In South Africa, the South African Radio Astronomy Service (SARAS) defined the threshold levels of harmful RFI across the entire frequency band (70 MHz to 25.25 GHz) 2.10. These threshold levels are more stringent than those in Australia, hence ideal for single-antenna observations.

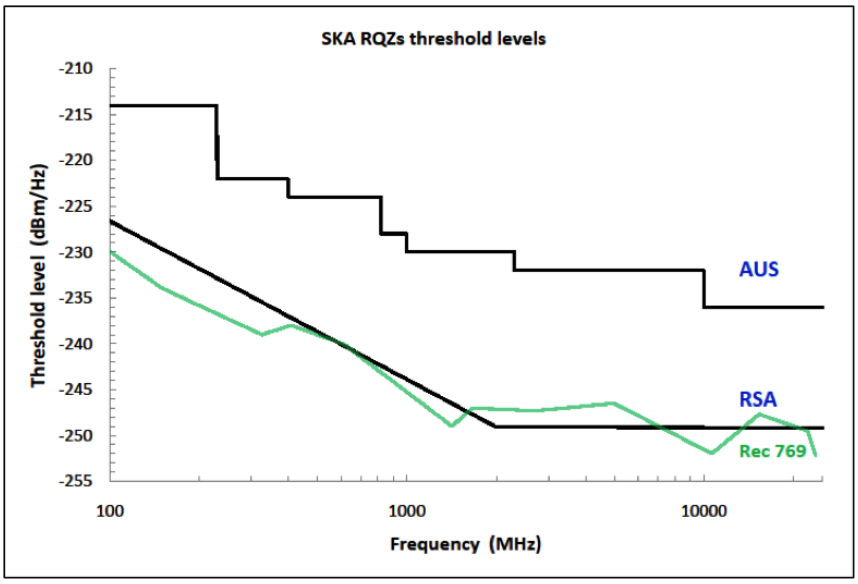



FIGURE 2.10: Threshold levels of harmful RFI listed in Australia (AUS) and South Africa (RSA). The line labeled Rec-769 is for frequency bands allocated to the radio astronomy service (RAS) under the ITU-R radio regulations [81].

## 2.5 The MeerKAT and SKA Radio Telescopes

The MeerKAT radio telescope in the South African Karoo is a precursor to the SKA project [53] and follows the Karoo Array Telescope (KAT7). The MeerKAT radio telescope consists of 64 antennas (Figure 2.11), each with a diameter of 13.5 meters (Figure 2.12). The frequency of the MeerKAT receivers ranges from 900 MHz to 1670 GHZ (Table 2.2) [53, 82]. Commissioning of MeerKAT was done in phases to allow verification of the system, early resolution of any technical issues, and initial science exploitation. MeerKAT is now fully online [83].

FIGURE 2.11: An overview of some of the 64 MeerKAT antennas (*Photo credit: SARAO* [82].

FIGURE 2.12: A single MeerKAT antenna (*Photo credit: SARAO* [82].

MeerKAT will be integrated into the international SKA telescope, which will be the largest and most sensitive radio telescope in the world. There will be two sites: one in the Northern Cape in South Africa (with remote stations throughout Africa) and the other in Australia [53]. The SKA is to be constructed in two phases.

Phase 1 (SKA1) in South Africa and Australia will provide 10% of the total collecting area [84]. The first phase of the SKA will consist of two arrays: SKA1-low will observe at low frequencies (50 MHz to 350 MHz). This array will comprise 512 stations spread over a large area (65 km), and each station will consist of 256 antennas, totaling over 130,000 antennas located in Australia. SKA1-mid will observe at mid-frequencies (350 MHz to 14 GHz) and will consist of 197 dishes (133 SKA dishes and 64 MeerKAT dishes) located in South Africa [85]. This array will extend to about 150 km, although the majority of the dishes would be concentrated at the core of the array [86].

Phase 2 (SKA2) will represent the largest capacity of the full array of over 2400 dishes, which will involve expansion into other African countries and wider areas of Australia. The second phase will include the addition of mid-frequency aperture arrays and expanding the Phase 1 arrays over larger areas. The full SKA will comprise about 3000 dishes covering thousands of kilometers (over 3000 km) to operate at frequencies up to about 10 GHz [87].

The image at the top-left of Figure 2.13 was the first light image produced in 2017 using 16 antennas [88]. This image shows a small patch of sky covering less than 0.01 percent of the entire celestial sphere, where MeerKAT revealed 20 times more galaxies than previously known, in this location. The second image (Figure 2.13 top-right), was produced in 2018 and shows the to-date most detailed view of the central region of our Galaxy in radio wavelengths. At the distance of the Galactic Center (within the white area near the image center), the 2-degree by 1-degree panorama corresponds to an area of approximately 1,000 light-years by

500 light-years. The image reveals never-before-seen features and a clearer view of known supernova remnants, star-forming regions and filaments [83].

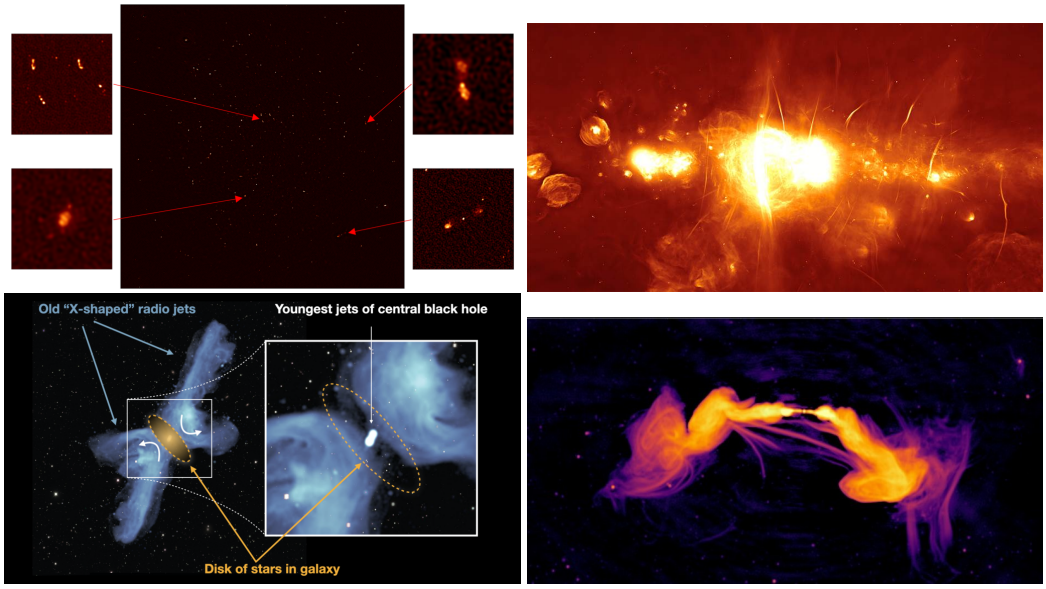



FIGURE 2.13: Top (left) – MeerKAT's First Light Image captured by 16 antennas in 2017 [88] (*Photo credit: SARAO*). Top (right) – So far the finest image of the center of the Milky Way galaxy captured by 64 MeerKAT dishes in 2018 [83] (*Photo credit: SARAO*). Bottom (left) – The image of the X-shaped giant radio galaxy PKS 2014-55 best described as a 'double boomerang' phenomenon, captured in 2020 [42] (*Photo credit: UP; NRAO/AUI/NSF; SARAO; DES*). Bottom (right) – The image that reveals new features in the distant galaxy (European Southern Observatory – ESO 137-006), produced in 2020 [89] (*Photo credit: Rhodes University/INAF/SARAO*).

The image at the bottom-left of Figure 2.13 was recorded in 2020, in the study of the X-shaped giant radio galaxy PKS 2014-55 [42]. This image indicates the old X-shaped radio jets, the younger jets closer to the central black hole, and the region of influence dominated by the central galaxy's stars and gas (disk of stars in the galaxy). The curved arrows denote the direction of the backflow of material that forms the horizontal components of the X. The MeerKAT observations enabled imaging of this source in unprecedented detail, which enabled the explanation of a previously blurry radio image of an 'X-shaped' galaxy as a double boomerang phenomenon [42]. The image at the bottom-right of Figure 2.13 is one of the latest images from MeerKAT captured in 2020. This image uncovers new features of the galaxy ESO 137-006. Ramatsoku et al. [89], describe these features as extremely collimated threads of radio emission connecting the lobes of the galaxy. The study further suggested that the radio emission from the threads is likely synchrotron radiation caused by the high-energy electrons spiraling in a magnetic field.

### 2.5.1 RFI Bands at the MeerKAT Radio Telescope

In South Africa, the regulatory authority body for the radio spectrum is known as the ICASA. ICASA monitors and regulates the use of the radio spectrum and the development of new radio applications. ICASA aligns its frequency allocations with those of the ITU under the ITU Radio Regulations (ITU-R RA.769-2) – protection

criteria used for radio astronomical measurements [12, 90, 6]. The ITU is the radio frequency spectrum controlling authority worldwide that ensures coordinated use of the spectrum around the world. The standardization of radio frequencies has enabled international projects, such as the Very-Long-Baseline Interferometry (VLBI), which has enabled astronomers to see the universe in greater detail than ever before [43]. The EM spectrum allocated to radio astronomy is limited, leaving the majority of the spectrum for commercial purposes [6] (Appendix A). Consequently, radio observations have been mainly affected by human-generated RFI. The TV and FM radio transmitters and GSM cellular networks emit high levels of RFI in the astronomical bands reserved for scientific observations, for instance, the MeerKAT bands (900 MHz to 1670 GHz) in South Africa (Table 2.2) [90].

TABLE 2.2: A summary of RFI bands at the MeerKAT radio telescope

| **Freq. Bands (MHz)** | **RFI Source** | **Description** |
|---|---|---|
| 925 - 960 | Terrestrial (Global System for Mobile Communications) GSM towers | Sporadic in both time and frequency occupancy. Will be reduced over time as alternative communications system is deployed. Currently 925 MHz to 935 MHz is unoccupied |
| 1085 - 1095 | Airborne (Secondary Surveillance Radar) SSR | Persistent, but variable, during hours of air traffic ($\sim$ 06h00 to 23h00) |
| 1082 - 1150 | Airborne DME interrogators | Sparse frequency occupancy ($\tilde{5}$ visible at any one time), narrow band (<1 MHz) signals, variable time occupancy during hours of air traffic |
| 1164 - 1300 | (Global Navigation Satellite System) GNSS Satellites | Entire band occupied continuously |
| 1467 - 1492 | WorldSpace Satellite | Strong and persistent. Does not occupy entire band. May be decommissioned in the future |
| 1525 - 1610 | GNSS and Inmarsat Satellite | Strong and persistent |
| 1616 - 1626.5 | Iridium Satellite, Geostationary Operational Environmental Satellite (GOES) | Persistent |

The frequency band from 925 MHz to 960 MHz is susceptible to terrestrial or ground-based GSM towers, usually sporadic in time and frequency occupancy. The frequency band from 1085 to 1095 MHz is susceptible to airborne-based RFI sources, e.g., SSR [3] radars and DME [4] interrogators. RFI from SSR radar is persistent and dependent on the air traffic period (06h:00 to 23h:00). RFI generated by the DME interrogator has sparse frequency occupancy, affects narrow-band, and depends on air traffic hours. The frequency band from 1164 MHz to 1626.5 MHz is susceptible to global telecommunication satellites (GNSS, Inmarsat, Iridium). Although some forms of satellite interference (e.g., WorldSpace) can be avoided, global geostationary and non-geostationary satellite transmissions are strong and persistent.

[3] SSR – Secondary Surveillance Radar.

[4] DME – Distance Measuring Equipment.

### 2.5.2 RFI Mitigation Pipeline at MeerKAT

In radio astronomy, RFI mitigation are techniques to remove RFI from radio observations. Several mitigation approaches have been introduced to reduce the damage on the desired signal in radio astronomy [10, 91, 9]. These are broadly classified as preventive (proactive) and reactive strategies, implemented at different stages of mitigation process[91]. Prevention is the first stage and involves modifying the local and regional radio frequency environment by setting up regulatory measures [9]. This is followed by the application reactive strategies tat occur within the receiver system. Common approaches include detection algorithms, physical filter systems and legislative actions [70, 92]. It is important to note that not all approaches are universally applicable to all radio telescopes, as each telescope has a unique physical, legislative and radio frequency environment [10, 93].

At MeerKAT, RFI monitoring is the first step in the mitigation pipeline. Human experts can successfully identify RFI through visual inspection, specifically by looking for "bright spots" (indicating large amplitudes) in waterfall plots, as exemplified in Figure 2.9. However, the vast amount of data has rendered manual identification impractical and infeasible. While explaining the detailed workings of automated pipelines falls outside the scope of this thesis, it is worth noting that various machine learning algorithms, including K-Nearest Neighbour (k-NN), Random Forest Classifier (RFC), and deep learning Convolutional Neural Networks (CNNs), have been utilized to some extent in mimicking the human visual inspection for RFI detection in radio astronomy [94, 95].

Monitoring at MeerKAT is implemented using two types of equipment: fixed monitoring systems (FMS), which provide continuous monitoring of the central site of the MeerKAT, and SKA Phase 1 mid-Frequency telescopes and mobile monitoring systems (MMS) that either reside on a vehicle or act as self-supporting platforms, which can be deployed anywhere within the telescope array and beyond [13].

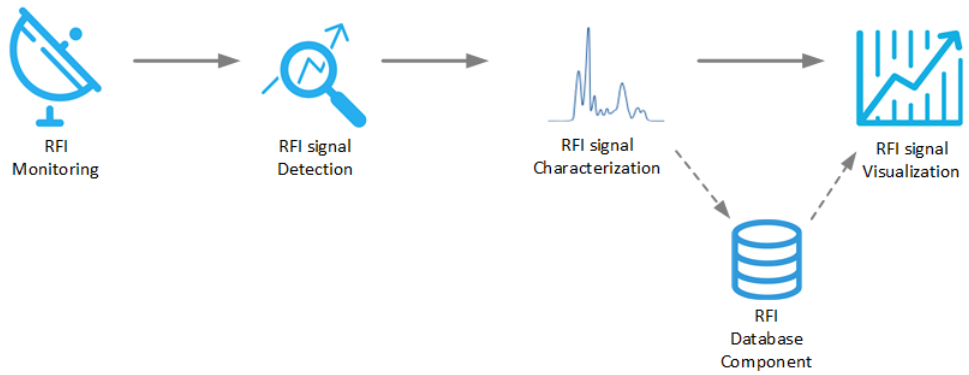



FIGURE 2.14: Showing the introduction of RFI database component in the current mitigation pipeline at the MeerKAT radio telescope.

The monitoring equipment provides access to, and visualization of, the spectral data collected in raw formats. At this stage, the RFI is detected, but not permanently removed [96]. Characterization of RFI signals is an essential step that follows the monitoring and detection stages [9], as it identifies easily a single RFI source based on the strength, geographical location or position of the source. However, factors, such as polarization, direction, orientation, periodicity over time, bandwidth, frequency distributions, modulation and encoding can make the process difficult [96, 69].

### 2.5.3 RFI Storage at MeerKAT/SKA

The RFI data is detected, measured and collected at the MeerKAT site in a range of different file formats, including images, text, documents, arrays and tables (Figure 2.15). The advantage of collecting RFI in different formats is to provide a subtle and detailed picture of the nature of the RFI source. However, the variety in RFI data formats has created a storage and access concern for the RFI data at the MeerKAT radio telescope. It is clear that working with multiple heterogeneous files limits data analysis and is also quite cumbersome.

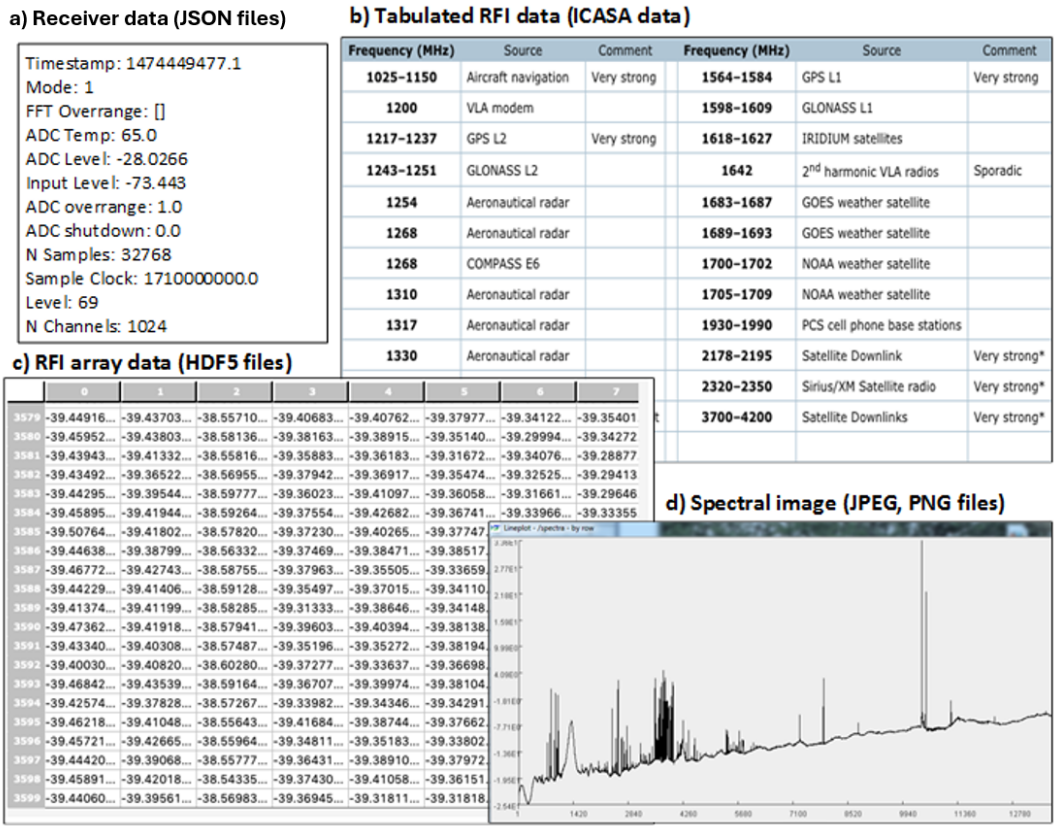



FIGURE 2.15: A sample RFI dataset in radio astronomy: a) receiver key-value data in JSON documents, b) tabulated list of licensed transmitters in spreadsheet, c) realtime analyzer (RTA) array data (3600 rows X 14700 columns) in HDF5 files, and d) spectrum image in jpeg/png showing RFI variation in specific radio frequency bands.

Data from the MeerKAT telescope is transferred from the Karoo processing building in Northern Cape to the Centre for High-Performance Computing (CHPC) in Cape Town [97]. It is thereafter transferred from CHPC to Inter-university Institute for Data Intensive Astronomy(IDIA) regional Science Data centers (at the Universities of Cape Town, the Western Cape, and Pretoria) [98] and released to scientists, researchers, and students for processing, analysis, and scientific discoveries [99]. The Science Data Processor (SDP) at IDIA processes about 4 TB/sec of MeerKAT data [100]. Large datasets of MeerKAT vary from 1 TB to 1.5 TB, while larger datasets are partitioned into 12 frequency ranges (i.e., 880-930 MHz, 930-980 MHz, 980-1030 MHz, 1030-1080 MHz, 1080-1130 MHz, 1130-1180 MHz, 1280-1330 MHz, 1330-1380 MHz, 1380-1430 MHz, 1430-1480 MHz, 1480-1530 MHz, 1630-1680 MHz) for efficient processing and storage [101].

The MeerKAT telescope uses HDF5 file format to store most of its data. Flexible Image Transport System (FITS) files are also used to store images or tables along with metadata but are not flexible enough for radio astronomy; hence astronomers resort to Measurement Sets (MS) or HDF5 datasets. While these files provide storage for astronomical data, they create artificial barriers in data, particularly during data analysis and integration. This is because measurement data is not mostly available in one file but rather stored in several files [17]. For instance, scientists who would like to retrieve and analyse data from other studies would need to write a new application program for each task [17, 20]. In addition, MS is unable to accommodate varying channel widths introduced by the baseline-dependent correlator, rendering it an incomplete solution for RFI storage. Therefore, these approaches are not sustainable, and thus, we suggest a database application.

Overall, RFI data collected in isolation has created a data integration problem. Moreover, large future volumes of data generated at high speeds during large survey observations at the SKA are likely to exacerbate the problem making storage and access excessively difficult. Such limitations have stimulated the development of innovative database models.

# Chapter 3

# Databases and Database Models

In this capter, we discuss database models suitable for storing RFI data, with a review of existing and emerging database models. Thereafter, we explore databases for scientific applications, focusing on scientific case studies. Lastly, we describe suitable techniques to evaluate these database models. An RFI database would store a list of all sensors, transmitters, locations and permit information. A permit is a document issued after thorough measurements of an emitting device or transmitter. This document details restrictions on the usage of the device at the telescope site. A database consists of data models that define how data is connected and how data can be processed and stored inside the database system. For example, an RFI data model would define the relationships among the RFI data objects, i.e., transmitters, sensors and locations, etc. Each data object consists of a unique data model responsible for determining how RFI data is stored and accessed from the database.

There are three main database models: the traditional relational database (Structured Query Language – SQL) model in which all data is represented in terms of tuples (rows of data) grounded into relations (tables) [102]; the non-relational (NoSQL) model in which data is modeled by means other than the tabular relations found in the relational databases [103, 104]; and the new-relational (NewSQL) model which conforms to the traditional relational principle but adds the scalability functionality [22].

## 3.1 Relational: SQL Model

Traditional RDBMS are regarded as SQL models primarily because they adopted SQL as their querying language [105, 102, 104]. In the relational model, the data is represented as a collection of relations linked together with quick and easy data retrieval (Figure 3.1). The relational model is dominant in the business sector (banking, insurance and e-commerce), whose top priority is to maintain reliable and consistent transactions. This is mainly because of the Atomicity, Consistency, Isolation and Durability (ACID – explained below) compliance property inbuilt in the relational model to ensure that financial transactions are completed correctly and securely. Relational models are supported by SQL, which is technically a standard for manipulating data in RDBMS. SQL syntax is simple, easy to learn and interpret. However, the highly structured nature of relational models is a drawback. Relational models use tables: a set of data (images, text, arrays, documents and videos) that do not fit in a table structure is difficult to accommodate.

ACID database properties are defined for conventional relational databases [106]. Atomicity guarantees that a transaction can be either successful or unsuccessful, but not both. Consistency guarantees that, in case of a transaction failure, the system reverts to the previous stable state, so that the system always remains stable. The isolation property guarantees that a transaction is completed

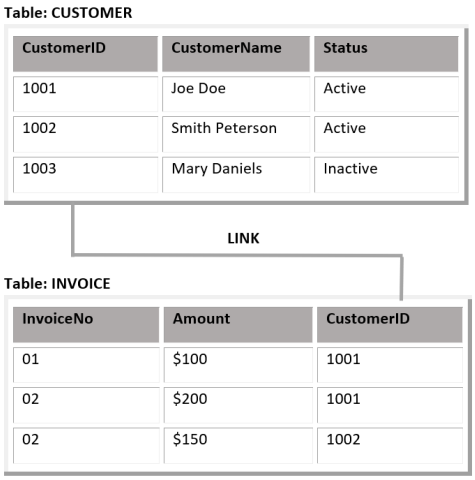



FIGURE 3.1: Illustration of two relations: *CUSTOMER* (parent entity) and *INVOICE* (dependent) linked via a defined relationship (*CustomerID*). A record of *CustomerID* (1001) is referenced twice in the *INVOICE* table; that is, *InvoiceNo* (01) valued at $100, and *InvocieNo* (02) valued at $200. Therefore *CustomerID* = 1001 with *CustomerName* "Joe Doe" has invoices amounting to $300.

without any interference and is processed independently. Lastly, the durability property guarantees that all committed transactions are saved in logs and will not be lost. This property helps in the recovery of transactions in case of abnormal terminations. Typical ACID databases are predominately characterized by consistent transactions. Essentially, use cases (financial, military and health care industry), whose transactions need to be consistent, are more appropriate in relational ACID databases [103, 102].

PostgreSQL [23] is a relational DBMS with many modern features, such as SQL sub-queries, multi-version concurrency control (guarantees consistent READs across the database) [107], as well streaming and replication, partly to accommodate the current database demands. In database systems, Multi-Version Concurrency Control (MVCC) is a technique responsible for concurrent control and transaction isolation, ensuring data consistency without relying on locks. MVCC permits each transaction to read a snapshot of the database at any given time, rather than accessing the persisted state of the data. This enables multiple transactions to read and write data simultaneously without interfering with one another. MVCC creates duplicate copies of records so that data can be safely read and updated at the same time. This technique ensures consistency in a database transaction as data is being read and updated simultaneously [107, 108]. A subquery is a query that is nested inside another query or subquery. Replication in databases refers to copying data from a primary database to one or more replica databases. Replicas not only ensure reliability during database system failovers but also speed up query processing.

## 3.2 Non-relational: NoSQL Model

NoSQL, also "not only SQL" model, does not store data in traditional table structure, but also provides other scalable means of storage including key-value, columnar, document-oriented and graphs discussed in more detail below [104]. NoSQL is classified as aggregate- or non aggregate-oriented. Aggregate-oriented models

comprise a collection of data objects considered as a unit of data or an aggregate. These database models (key-value, columnar and document-oriented) collect data from nodes in a cluster and then re-arrange it into different aggregate formats. This approach allows users to operate on data in units that have a more complex structure, where each unit can have a specific data structure. This is useful when the same aggregate is used frequently within the database cluster. Non-aggregate models (graph databases – see below), on the other hand, are a superset of aggregate-oriented that rely heavily on relations of a limited data structure. Both models are schema-less – no predefined data structures, while the only difference is that an aggregated model splits relations to operate on specific aggregates in a cluster [109].

The NoSQL database model supports web applications for which relational database models are not well-suited [102]: these are applications characterized by a data structure that does not fit well into relational tables: more READs than WRITEs, a high degree of scalability, availability, and performance, and a certain level of consistency [110, 111, 112]. High performance means it supports many users' queries, and high availability means that the database remains available, even during node failure within the cluster. Scalability means that both the data and database operations are evenly distributed across several nodes [22, 113]. NoSQL database models sacrifice some level of consistency (Basic availability, Soft-state, and Eventual consistency – BASE compliance) to achieve high availability and performance [106, 113].

BASE properties are defined to support NoSQL applications, such as social networks, wikis, blogs, and large-scale applications [106, 114]. Basic availability ensures the apparent availability of the data, implies that when a single node fails, the data is partially inaccessible, but the rest of the data layers remain functional. The soft-state property allows for temporary inconsistencies within the database. This means that during concurrent updates, partial updates can cause replicas of the same data item to hold different values. Nevertheless, the system eventually achieves consistency by synchronizing all replicas to the same value. The eventual consistency means that transactions are not updated instantly, but are propagated subsequently to all nodes.

### 3.2.1 Key-Value Model

The key-value stores (dictionaries or hash tables) store data as keys with their associated values [115]. A unique key is given to every object inserted into the database and is ordered for faster access. Data access is by primary key only, and a hash function is used to calculate the location of the data given a key. The data is usually semi-structured or unstructured, ranging from scalar data types (integers) to complex structures, such as JSON, Lists or BLOB (Binary Large Objects – image, video). As a result of its data model, key-value stores have a flexible schema and are highly scalable. Examples of key-value NoSQL models include Voldemort, Redis and Riak [104, 113].

### 3.2.2 Column-oriented Model

Columnar (also Column-oriented or Column-Family) stores are based on the Google Bigtable architecture [116]. Here, data is stored in cells of columns and grouped into column families, which is particularly essential for data organization and partitioning. The concept is similar to that of row-oriented relational databases.

However, the Column-Family follows a column orientation (writes data to disk by column), in which data is processed by each column using a key index distributed in multiple nodes [117]. The columnar database model performs I/O operations only on the sections of the disks corresponding to columns being read or updated. This leads to effective use of memory since only a portion of the data is accessed instead of the entire data block [109]. Moreover, it is also faster as most tables contain several columns, which are rarely used simultaneously by queries. Columnar databases are best suited to queries that involve a single column or column family. While columnar stores allow for the aggregation of data, it is not efficient when queries need to access multiple columns that do not belong to the same family.

Columnar databases are less efficient for Writes compared to row-oriented databases. When inserting a new record into a columnar database, all the column files need to be rewritten because the data needs to be inserted in the middle of a sorted table. On the other hand, in a row-oriented database, inserting a new record only requires writing the data at the end of the current data since the data is written in sequential order [113, 117]. Furthermore, columnar databases lack standard query notation, resulting in queries that consist of several lines of commands.

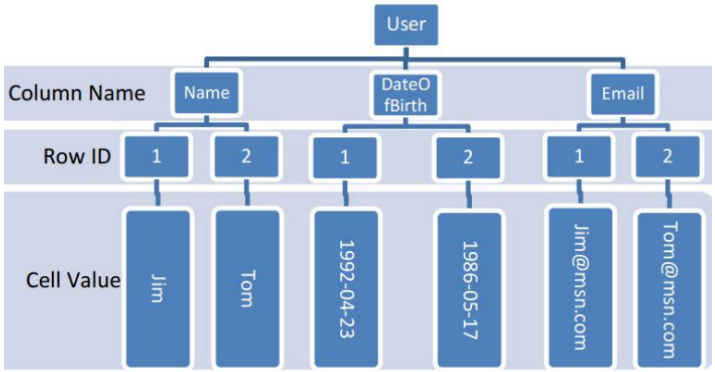



FIGURE 3.2: Columnar model showing user information (UserID, name, DateOfBirth, Email) grouped by attribute and stored as columns in multiple locations on the disk. All usernames are stored together as columns in one partition, all Dates of Birth stored together as columns in another partition [112].

Examples of column-oriented databases include Cassandra [118], HBase (Hadoop) [119] and Accumulo [117]. Unlike Cassandra, HBase and Accumulo database runs on top of the Hadoop Distributed File System (HDFS)[120] in the Hadoop Stack. HBase in particular, works entirely with HDFS, which is associated with several performance overheads. HDFS is responsible for organizing data storage on each node into a single, large and redundant file system. For Accumulo to achieve a good performance, the data and operations should be uniform across the cluster [121].

### 3.2.3 Document-oriented Model

In document-oriented Databases, the data is represented in JSON format [122]. A set of documents is called a 'collection'. Each document has attribute metadata listing the fundamental data type, such as dates, arrays, numbers, strings, binary data, or a sub-document. Data can be indexed and queried based on the attributes [123, 124]. Collections in document-oriented databases are suitable for storing large and unstructured datasets due to their dynamic schema (without predefined schemas):

documents within documents (nested documents) and arrays of documents. Nested documents are an efficient way to store related data, particularly when the data is frequently accessed together. However, this may cause an impact on Reads as data volumes increase in a document. There is no strict schema that documents have to conform to, thereby eliminating the need for schema migration efforts [113]. Schema migration involves translating data from one schema to another, which may affect the data quality. MongoDB [125] is one of the popular document-oriented database systems.

### 3.2.4 Graph Model

Graph databases store and display data, using nodes and relationships between the data. The nodes and edges can have any number of properties associated with them. Nodes and edges can be analogized to the entities and relationships in relational databases. Additionally, they possess the capability to ensure ACID transaction properties. Graph databases hold the relationships between data as a priority. As a result, this leads to fast queries. There can be more than one relationship between two nodes. A graph database model is suited to applications requiring a need to query the relationship between entities and their properties. These applications may include location and navigation-based services, network and IT infrastructure, fraud detection, metadata management and social networking (Facebook, Twitter and Dating Apps) [124, 113]. Neo4j [126] is an example of a graph database system.

In summary, the strength of the graph databases lies in the shortest path (distance of one node to another), networked data, and other relationship-based applications. However, they lack the stability to work in a distributed cluster [113]. Key-value stores have fast lookups, although the store lacks a schema [115]. Document stores are effective in storing sparse data. However, they are slow at queries requiring fetching data from a cluster [113, 124]. The column-oriented databases have fast lookups and great performance in a distributed environment. On the other hand, they have a limitation in query notation or syntax [112, 121]. Generally, NoSQL employs denormalization, aggregation and secondary indexing [117, 127]. Database normalization involves removing redundancy (duplicate data), so only a single copy exists of each data. Normalization reduces inconsistency and supports data integrity. Denormalization, on the other hand, adds redundant data to the existing data to improve the read performance of the database. Besides relational databases, data in the NoSQL database is denormalized to provide for high availability and scalability. Therefore, this makes NoSQL thrive in a distributed environment more than SQL databases [128].

## 3.3 New-relational databases: NewSQL Model

NewSQL or the modern relational model is designed to conserve the properties of the traditional relational model while incorporating some properties of the NoSQL model, such as availability and scalability [104]. Here, the data is managed in several ways to ensure data integrity and consistency, while considering fault tolerance in the database cluster. Thus, several NewSQL databases provide horizontal and vertical scalability to achieve full functionality: horizontal scaling relates to increasing the number of commodity nodes (hardware) in the cluster, whereas

vertical scaling involves enhancing the CPU and RAM capabilities of the commodity hardware [22].

NewSQL has been used in a range of scientific applications, such as astronomy, oceanography, fusion, remote sensing, climate modeling and seismology [21]. Scientific data have the natural structure of an array [22].

| SensorID | Step 1 | ... | Step 500 | ... | Step 5,000 |
|---|---|---|---|---|---|
| "A" | (12.5, 75.1, clear) | ... | (11.5, 69.5, cloud) | ... | (9.5, 65.2, rain) |
| "B" | (1.0, 55.2, cloud) | ... | (0.5, 55.2, clear) | ... | (2.0, 55.2, clear) |
| "C" | (10, 35.1, snow) | ... | (12.5, 34.2, snow) | ... | (12.5, 33.8, snow) |
| "D" | (0.5, 85.1, clear) | ... | (1.0, 85.5, clear) | ... | (0.5, 85.7, clear) |
| "E" | (15.2, 66.2, rain) | ... | (7.9, 66.5, clear) | ... | (12.7, 66.9, clear) |

FIGURE 3.3: Array database model composed of connected cells. Each cell in an array contains a vector of data values (a composite data type) [21].

The dimensions of an array can be user-defined data types, such as strings, floats, latitude and longitude. A basic array can store and manage larger files of higher dimensions; for instance, a data file of wind speed, temperature and conditions, each with 5000 iterations captured from five sensors over a specific period (Figure 3.3) [21]. SciDB is a new relational DBMS with Postgres-style user-defined functions (UDF). SciDB has a native data model with data stored as n-dimensional arrays and not fixed 2-dimensions (tables) found in relational databases. UDFs can support ad hoc queries that are common in scientific discoveries. SciDB also supports most functionalities found in PostgreSQL [129].

## 3.4 Polystore Model

The BigDAWG (Big Data Working Group) [130, 131] polystore model has storage engines or data stores that are distinct and accessed separately through their query engine. The data stores are heterogeneous: stores that can handle data of different formats. The polystore approach uses several data models (SQL, NoSQL and NewSQL), languages, and store engines to yield a significant performance advantage (Table 3.1) [132, 133], through polyglot persistence: using many data stores for different scenarios or application needs [134, 135]).

TABLE 3.1: Comparison of performance metrics for current database models.

| | SQL | NoSQL | NewSQL | Polystore |
|---|---|---|---|---|
| Example | PostgreSQL | Accumulo | SciDB | BigDAWG |
| Application | Transactions | Search | Analysis | All |
| Data Model | Relational Tables | Key-Value Pairs | Sparse Matrices | Associative Arrays |
| Math | Set Theory | Graph Theory | Linear Algebra | Associative Algebra |
| Consistency | X | | | X |
| Volume | | X | X | X |
| Velocity | | X | X | X |
| Variety | | X | | X |
| Analytics | | | X | X |
| Usability | X | | | X |

Table 3.1 compares the capabilities of store technologies. The polystore approach combines each store's capability to enhance its full functionality. For example, the SQL database model cannot handle large volumes of data, high-velocity data and detailed analytics, but guarantees strict consistency on both Reads and Writes. Consistency means that data must align with all data points in the database during Read and Write operations. This property helps SQL to ensure data consistency. NoSQL does not have strong consistency (but rather offers weak or eventual consistency). Weak consistency means that the changes made are not instant but ultimately come into effect. NoSQL store has a procedural query language but provides high performance for very large data volumes, high velocity, variety and detailed analytics. NewSQL cannot ensure consistency, variety and usability but guarantees support for large volumes, high velocity and deep analytics. The polystore provides for most storage demands through the combination of capabilities across different stores [130, 136, 131].

Several databases are integrated; but each maintains application-specific properties, integrity control and safety control [137, 138, 139]. Each database system requires independent operation environments, database engines [1], data structures, and semantics – and must also maintain the accuracy of a query during schema translation as data moves from one system to another. The polystore model has an integrated middleware that coordinates the isolated database environments, distinct data models, languages and engines [136, 131, 130]. The middleware enables the polystore to match and query data across multiple data stores.

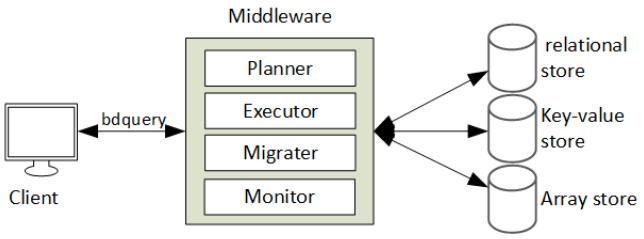



FIGURE 3.4: Basic polystore database framework [136]

[1]The engine in a DBMS is a core component of the database system used to create, read, update, and delete data in a database.

Figure 3.4 shows a basic polystore framework with the four components of the middleware that coordinate operations across engines: the planner, executor, migrator, and monitor [133]. Incoming queries may interact with one or more of the underlying storage systems based on the query characteristics. For instance, a linear algebraic query operation on time-series data may utilize only the array database, while a join operation between time-series data and catalog data may access both the array and relational databases. Join operations involve data stored in two or more places or data stores. To do this, the planner parses an incoming query into a collection of data objects to identify possible query plan trees. The query plans are then sent to the monitor to determine the ideal engine for execution based on the experience of related queries. The executor determines an optimal method from a combination of operations with the help of a migrator that moves objects within engines or islands, once required by the query plan [140, 130].

The polystore model includes islands, shims, and casts. The islands provide information on data models. The casts facilitate the migration of data between engines, and the shims facilitate the translation of data from one data format model to another. The data may be stored in one file or a directory, or via some other mechanism that defines the scope of data used by a group of applications. Each database creates an island of information about itself, even if the database is partitioned and distributed over multiple machines [131]. Location independence in polystore allows users to operate on the data without explicit knowledge of the location, thus reducing the overheads associated with data transfer across multiple storages. Semantic completeness supports a wide range of complex queries [131, 136].

## 3.5 DBMS Case Studies

In this section, we discuss case studies of databases and database systems that have been used in different scientific applications.

### 3.5.1 PostgresSQL

PostgresSQL has been used in astronomy to support several storages and cross-matching tools, such as VizieR, SIMBAD, and TOPCAT, to improve the capability of handling astronomical data [24]. Cross-matching in astronomy involves searching an entry of a source in one catalog corresponding to another catalog using the source location as a focal point [24]. VizieR [141] provides access to astronomical catalogs and data tables online. These tables are created in Postgres and the description links to physical and astronomical sources found in the reference catalog [142]. The tables are available via a file transfer protocol (FTP) service that enables users to browse data tables to discover and extract information. TOPCAT (Tool for OPerations on Catalogues And Tables) [143] is an interactive graphical viewer and editor for tabulated data. It offers numerous data views: a browser for the cell data, information about tables and columns (metadata), an interactive higher-dimensional (3D) visualization, and computing statistics, as well as joining tables using flexible cross-matching algorithms. Cross-matching tables from large surveys result in large tables. PostgreSQL is used in a 3D (XYZ coordinate system) as the database index and applies other techniques to enable high-speed searches and cross-matching huge catalogs. SIMBAD [144] is a reference database that offers cross-identifications, bibliography, and measurements of astronomical objects

outside the solar system [145]. SIMBAD has a variety of query modes, i.e., object name, coordinates, and various criteria. The Postgres queries in SIMBAD and VizieR may consist of small lists of objects and scripts.

These tools do not support large catalogs during cross-matching tasks. However, the extensible features of PostgresSQL have allowed scientists to create user-oriented cross-match tools such as QuadTree Cube (Q3C) [24] used to improve the quality of identification continuously and increase the number of cross-matching records to tens of millions of lines.

### 3.5.2 SciDB

SciDB can store data in multi-dimensional arrays. It automatically distributes and computationally loads data across several nodes while providing a unified system view to a user [146]. Not only do multi-dimensional arrays provide a natural structure for scientific data, but they also allow the construction of an ever-growing library of efficient mathematical operators [147]. These libraries can enhance SciDB's functionality through the use of user-defined types. SciDB outperforms not only traditional DBMS, but also other large-scale tools used for data storage.

EarthDB [25, 148] is a database project based on SciDB that focuses on analyzing NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) data [148]. The MODIS data consists of imagery obtained in 36 spectral bands, encompassing visible and infrared from solar and thermal radiation. These images are captured at three distinct spatial resolutions: 250 m, 500 m, and 1 km pixels. Unsurprising aspect is that a MODIS swath can cover the entire Earth within a single day, spanning a width of 2330 km [25, 148]. The collected MODIS data is divided into granules ranging from 64 MB to 10 MB at every 5-minute intervals during orbit time. These volumes accumulate when data is data gathered for long-term (seasonal, annual, decadal) analysis, which poses significant challenges in terms of storage and analysis capabilities [25]. EarthDB uses the SciDB multi-dimensional array database to analyze MODIS, level 1 satellite data from NASA [25, 149]. MODIS data is sourced from orbiting satellites (Terra MODIS and Aqua MODIS) that scan the entire surface of the Earth to understand global dynamics and processes occurring on the land, in the oceans, and the lower atmosphere.

EarthDB imports MODIS data into SciDB. EarthDB consists of a preprocessor and parallel loader component for importing large datasets. The preprocessor extracts MODIS HDF-EOS (HDF- Earth Observing Systems) data sample, and the parallel loader increases the ingest rate of the system by taking advantage of SciDB's ability to perform distributed loading from several nodes. EarthDB has multidimensional schemas that represent MODIS data as unified storage. EarthDB uses a high-level query language for fast filtering and analysis of MODIS data. EarthDB's capability to define and reconfigure the entire data analysis pipelines, using the SciDB database, has enabled rapid ad hoc analysis [25, 149].

### 3.5.3 Polystore

The MIMIC-II medical dataset [150] health records for thousands of critical care patients, including patient metadata, free text-form data (such as notes taken by medical professionals), semi-structured data (such as prescriptions and lab results), and waveform data (measurements from bedside devices such as heart rate, pulse). MIMIC-II is a polystore that uses many stores of different types to manage heterogeneous data in a medical environment [130, 136]. Here, PSQL stores all

patients' metadata in well-structured tables. SciDB is used to compute and store the FFT of a patient's waveform data and compare it to what is considered normal. Accumulo is used for text searches, such as looking up a patient's notes or text from medical professionals.

Polystores have been used for ocean metagenomics data [151], which contains data on bacteria found in seawater samples. This dataset comprises sample and sensor metadata, genome sequences of about 20 million sequences per sample, cruise reports, and streaming sea flow data. They use data stores (Accumulo, PSQL, S-Store and SciDB) [131] to optimize different data applications. PSQL stores structured data such as sensor and sample metadata and historical data collected from the SeaFlow instrument. Accumulo is used to store free-form text reports and pieces of the genomics dataset. S-Store is used for recording and processing streaming SeaFlow data. SciDB stores and processes components of the genomic data [133].

Each store of the polystore is used to optimize different applications (exploration, navigation, text and geo-analytics, and heavy analytics) on the metagenomic data [133, 152]. For instance, PSQL gives researchers a quick look at the overall dataset. The navigation application uses streaming data from SeaFlow to reduce the cost of data collection cruises. Here, a researcher enters a parameter of interest in the system and uses historical tracks stored in PSQL and real-time streaming tracks in S-Store to offer navigational suggestions. Text and geo-analytics application help researchers look for details about cruise stations and search cruise logs within a particular region stored in Accumulo and PSQL. Heavy analytics applicant provides researchers with analytics that cut across metadata and genomics data. This application looks for metadata stored in PSQL and genomic data migrated from SciDB. The spatial predictive models for analyzing track data make use of PSQL and SciDB [133].

## 3.6 Database Evaluation

Databases are benchmarked by executing a specific workload against a given dataset. Benchmarks are vendor-specific, which means users can compare performance or select a system based on specific vendor target workloads, configurations or infrastructure, but comparison across different vendor systems is not possible [153, 154]. Therefore, new database benchmarks have emerged for scalable databases (such as ScalableSQL, NewSQL and NoSQL).

### 3.6.1 Yahoo! Cloud Serving Benchmark (YCSB)

YCSB [155, 156] is a benchmarking methodology proposed by Yahoo for the performance evaluation of scalable databases. The performance of the benchmark has been tested on four widely used systems: Apache HBase, Apache Cassandra, Yahoo!'s PNUTS, and a distributed MySQL implementation [157]. Studies [103] show YCSB as the best benchmark for evaluating scalable databases. The benchmark measures latency characteristics as the server load increase [157]. YCSB++ is an extension of YCSB that supports multiple-phase workload definitions and coordination of several clients to increase the load on the database server [156].

YCSB runs on several platforms and supports a wide range of database applications [157, 158, 159]. The YCSB benchmark consists of two components: a generator of data and a set of performance tests to assess Read and Update operations. Each test scenario is regarded as a workload defined by either a

percentage of Read and Update operations, total number of transactions or number of records [160]. Table 3.2 shows a list of YCSB default workloads.

TABLE 3.2: YCSB Workloads

| Workloads | Operations (sec) |
|---|---|
| A – Update Heavy | Read: 50% Update: 50% |
| B – Read Mostly | Read: 95% Update: 5% |
| C – Read Only | Read: 100% |
| D – Read Latest | Read: 95% Insert: 5% |
| E – Short Ranges | Scan: 95% Insert: 5% |
| F – Read-Modify-Write | Read: 50% Read-Modify-Write: 50% |

Workload A (also regarded as 'Update Heavy') consists of 50% Read and 50% Update operations; Workload B ('Read Mostly') consists of 95% Read and 5% Update operations; Workload C ('Read Only') constitutes 100% Read operations; Workload D ('Read Latest') consists of 95% Read (newly inserted inclusive) and 5% Insert operations. In this workload, the newly-inserted records are treated as the most recent Reads. Workload E ('Short Ranges') consists of 95% Scan and 5% Insert operations. This workload involves a small range of records (e.g., from 1 to 100 records) instead of considering a single record. It also involves 5% insertion of new records. Lastly, Workload F ('Read-Modify-Write') involves 50% Read and 50% Read-Modify-Write operations. In this workload, a record is read and updated, and all the modifications are written or saved. Other workloads extended to include heavy Update operations include Workload G (5% Reads and 95% Updates) and H (100% Updates only) [161].

### 3.6.2 Database Response Time and Speed

Database response time, latency and speed are common measurement parameters used in database evaluation. Database response time refers to the time the database takes to return the result – also known as the total execution time taken upon a query completion. The response time varies depending on the database and the optimization mechanisms. Most scalable databases use volatile memory to reduce the overall response time, though this is more costly [161]. Volatile memory, such as cache memory, loses data when the device is not powered. The databases using volatile storage experience shorter execution times due to the fast speeds of volatile memory compared to the speed of retrieving files stored on the hard disk [160, 162]. Some of the databases with the fastest response times experience between 4 to 20 seconds while fetching bulk workloads of about 10,000 and 100,000 records [163, 164]. Response time is a key parameter in determining the speed of the database, although it often depends on the execution machines. The shorter the execution, the faster the speed of retrieval. Due to the rapid proliferation of new and scalable database systems that differ greatly in applications and workload, there has been a standardization challenge. Therefore, studies have not decided on the standard acceptable values for download speed and response time [102, 160, 162].

### 3.6.3 Database Latency

In databases, latency refers to the delay for the data to become available for download from the database. Database latency is different from query latency which

refers to the time it takes to execute a query and receive the results [165]. Database latency forms a small component of the total response time. The database latency is caused by several factors, such as distance, propagation delay, internet connection, data content, wireless network, the router and other networking devices [166]. In other words, latency in databases is due to mostly delays at the server and the network path.

Database latency determines the ability of the database to handle realtime applications. Real-time databases must download as soon as the query transaction occurs (zero latency). In contrast, near-realtime requires the data to download after a set delay interval. The acceptable minimum latency without interruption at the database server is one second or less [167, 168, 157]. Beyond the maximum acceptable latency (> 1 second), the database is likely to experience a high variance in latency. This is undesirable as high variance will impact the overall response time of the database, consequently affecting the performance of the database [167]. On the other hand, low and stable latency is generally acceptable and most suited for real-time and near-time applications such as the onsite RFI monitoring and detection at the MeerKAT/SKA telescope.

In conclusion, several database models have been developed to match the current data needs. However, each database model serves specific applications. It is, therefore, imperative for our study to choose the right database models suitable for RFI storage and monitoring at the MeerKAT/SKA radio telescope.

# Chapter 4

# Design

In this chapter, we aim to design a scalable database model that can store RFI data in diverse formats. Our approach has four key phases: requirements analysis, database design (which involves three successive design stages, i.e., conceptual, logical, and physical), implementation, and evaluation (Figure 4.1).

We begin the design process by gathering user requirements. This is followed by creating the database model using the three fundamentals of database design: conceptual, logical, and physical designs [169, 170]. We create the conceptual design, followed by the logical and physical design using a polystore architecture. We also present an alternative design for storing RFI data within the same architecture. Subsequently, we discuss the structure of the polystore. Figure 4.1 illustrates

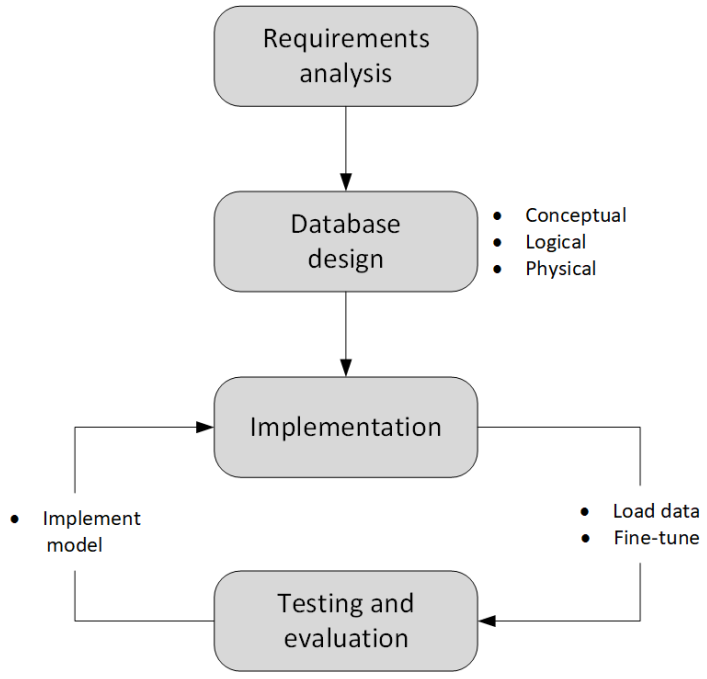



FIGURE 4.1: Phases of database design and implementation [171, 172]

the database design phase following a successful requirements analysis. The implementation and evaluation phases are carried out concurrently in an iterative fashion. The approach then follows the fundamentals of database designs [171, 169]. Our design philosophy is to leave data in its native format to avoid unnecessary translations that may lead to a loss in data quality.

## 4.1 Requirements Analysis

Over the course of three weeks, we gathered users' requirements from a group of 23 participants that included radio astronomers, engineers, and computer scientists. The data was organized in Google spreadsheets with an electronic questionnaire formulated with the help of Google Forms (free online survey tool) [173]. The purpose of the questionnaire was to investigate the nature of RFI data and how the RFI database would be used. Therefore, our questions were structured mainly around RFI data sources, RFI data types, and usage of the RFI database (see Appendix B).

The questionnaire was emailed to the RFI Working Group mailing list that included scientists from SARAO, South African Astronomical Observatory (SAAO), University of Cape Town (UCT), University of the Western Cape (UWC) and Rhodes University. We collated all responses and grouped similar responses under one requirement. These requirements were then mapped onto the database components necessary to support them (Table 4.1).

TABLE 4.1: RFI database user requirements and corresponding database design components.

| User requirements | Database components |
|---|---|
| 1. Input ad hoc queries<br>2. Load raw files | Standard API |
| 3. Store metadata<br>4. Store monitoring data<br>5. Store archive and reports | Integrated RFI storage |
| 6. Discover unknown RFI<br>7. Flag and update RFIs | Rapid RFI identification |
| 8. Generate occupancy plots<br>9. Compare new vs. old RFI tests<br>10. Plan measurements and observations | Timely statistics |

Meeting all the user requirements thus required four design components: a basic interface – users required the database to have a web-based platform to provide uniform access; RFI storage – they required the database to store and integrate all RFI data efficiently; RFI identification – they needed to plug into the database detection algorithms to detect RFI; and timely statistics for which users required to generate RFI plots and aggregated data.

The first two design components had to be achieved first to support the other two. However, according to the preliminary analysis, the data showed that about 80% of the respondents rated RFI storage and RFI identification as important requirements. We argued that for RFI identification to be successful, RFI data needs to be stored efficiently in the database. We, therefore, focused our work on creating a scalable RFI database. Future researchers can focus their efforts on other requirements.

### 4.1.1 Interface

A basic web-based interface is required for Reads and Writes operations. The interface should preferably support a uniform structure-like query language, such

as SQL. It should also be able to upload large data files and run ad hoc queries. We suggest a basic interface (Appendix J) because it is not the core focus of this work.

### 4.1.2 RFI Data Storage

We focus on this aspect of the RFI requirements. The database should be able to store and quickly retrieve RFI data that have been collected using different devices (telescope, sensor, RTA). RFI data include time and frequency data, archives (historical RFI data), and metadata (data dictionaries, schema description, and system information). A data dictionary stores data that describes each data object used in the database cluster. We create a schema catalog to store the schema information of each data object. The system table has information on DBMSs used within the cluster. These tables are linked and can be viewed as a single table. The database should act as a repository into which new RFI data is uploaded.

The database must integrate RFI data sourced from several devices into a uniform repository that provides easy access to data in different formats. The database should also scale to provide support for varying storage needs. We emphasize storing data in its native formats to avoid unnecessary schema translations.

### 4.1.3 RFI Identification

Users require both real-time and offline processing of RFI. This means the database should support identifying RFI during or after an observation. The database should plug in monitoring and detection algorithms to speed up the identification process which involves tasks such as cross-matching unknown RFI with the RFI database, adding permits to known RFI, flagging RFI, and updating the RFI database. RFI identification is not the focus of this work as it is a later process that will be built on top of a working database.

### 4.1.4 RFI Statistics

Users require graphs such as RFI occupancy plots which show different levels of RFI over a given period (i.e., daily, weekly, or monthly). The database should provide searchable statistics to guide scientists during observational measurements, equipment designs, and issuing RFI permits. Not the focus, as RFI data needs to be stored first in the database before generating statistics.

## 4.2 Designing the RFI Database

We followed a top-down database design strategy [171, 169], first identifying the data and then defining the data objects. The data objects were then allocated to separate datasets based on their structure. The data structure dictates the appropriate data object and hence the relevant data store. For instance, relational data that includes all RFI database metadata is stored in relational tables. Key-value data containing RFI measurement data, text, and RFI reports is stored as key-value pairs (JSON documents). Array data, such as RFI scans and multi-dimensional data (3D frequency, time, and polarization data), is stored as arrays. We chose three distinct data stores for our RFI database, each tailored to the data structure it suits best. Specifically, PSQL is selected to store relational data [24], Accumulo to store key-value data [117], and SciDB for array data [21].

Our database design process consists of three stages: conceptual, logical, and physical [171, 169], to transform the conceptual representation of the diverse RFI dataset into a unified physical database design.

The initial stage is conceptualization [174], aiming to capture the core content of the RFI database without considering storage structure. This phase employs techniques like entity-relationship modeling (ERM) [175, 176]. ERM is a modeling technique that involves visual representation or diagram (Entity-Relationship Diagram - ERD) utilized to illustrate relationships among different data objects (entities) within a database. It outlines how data is organized and interconnected.

Moving to the logical stage, we develop a logical database design from the conceptual design. This involves defining the data model in detail, including relationships between attributes, tables (data objects), and data entries. We employ the Unified Modeling Language (UML) notation [177, 178] to our ERDs to precisely represent the data objects (entities), attributes, relationship constraints, and entity behavior. UML notation serves as a standardized means to model and visualize various database design aspects in detail compared to other notations (Chen, or Crow's foot notations) [178, 179]. We present the ERD in UML notation, which is then reduced to the logical schema of the database. Subsequently, we review the logical schema to ensure the appropriate design of attributes and relationships, as well as an accurate representation of data integrity within the database. This step is crucial to ensure that the logical design encounters no difficulties in the implementation.

The last stage is the physical stage, in which we create a physical database design based on the logical schemas. This stage describes the storage structure as implemented within a particular database system, encompassing low-level structures (or physical schemas) for tables and attributes with comprehensive details such as data types, and indexing strategies. Essentially, the physical model outlines how data is stored and accessed, with the specifics dependent on the type of database system and hardware employed [180].

### 4.2.1 Conceptual Design

During the conceptual design stage, we established the main data objects suitable for storing the RFI dataset. We present four key objects: RECEIVER, RFIEVENT, TRANSMITTER, and PERMIT. The RECEIVER object stores data from the receiver device that captures the RFI signal, while the TRANSMITTER object collects data from the transmitter device that emits RFI signals. An RFIEVENT object stores all RFI occurrences or events captured during observations, and a PERMIT object stores documents or permits authorized by SKA engineers after thorough measurements on a device emitting RFI. The RFI permit outlines strict conditions for a device's use at the telescope site. Each data object can encompass sub-objects, resulting in an object with multiple fields.

We identified relationships between objects using ERM and represented the conceptual model in an ERD. Figure 4.2 is an ERD showing a conceptual model of the RFI database.

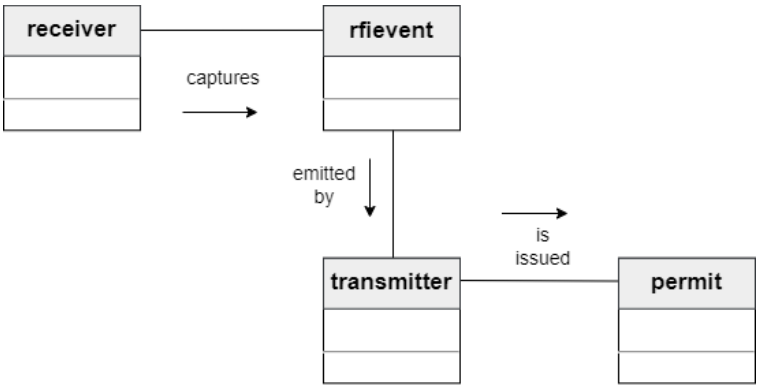



FIGURE 4.2: Conceptual model of the RFI database.

The connection lines with labels signify relationships between objects, and rectangles with internal labels symbolize entities or data objects. Our model's relationships are named "captures," "emitted by," and "is issued," derived from the connections established between objects. For instance, a receiver captures RFI events, RFI events are emitted by a transmitter, and a transmitter is issued a permit.

Overall, the conceptual model focuses on the entire RFI dataset to create a database structure or model that emphasizes the organization and associations between the data.

### 4.2.2 Logical Design

We translated the conceptual model by defining the data model, including data objects and their attributes, relationships of each data object, and modeling or design constraints to be enforced on the data. We represent the logical model using an ERD in UML notation. Afterward, we translate the ERD into a logical schema (Section 4.2.3). Figure 4.3 displays an ERD of the RFI database, which indicates each data object and its attributes (see Section 4.2.3 for complete attributes), along with relationship constraints when one object interacts with another.

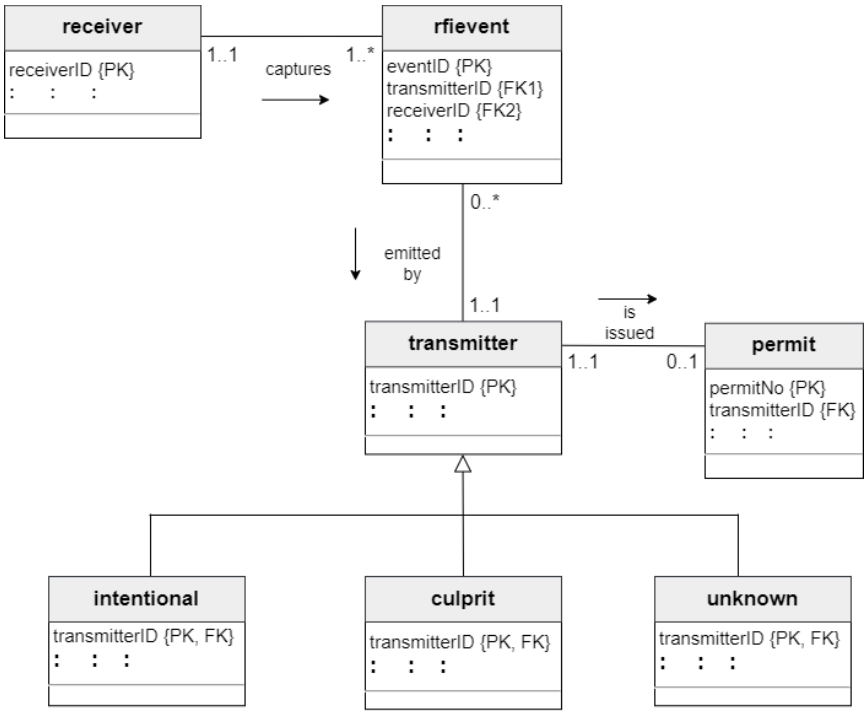



FIGURE 4.3: ERD of the RFI database (Refer to the logical schema showing all attributes in each data object (Section 4.2.3)

We show the four main objects, the same as in conceptual design. However, we introduce three sub-objects under the transmitter object that include intentional, culprit, and unknown. The intentional object stores data from the intentional transmitters, which are transmitters licensed nationally and can legally transmit (except in the radio-quiet zone). A culprit object stores data from devices or transmitters that have been detected at the MeerKAT site. We refer to these as "culprits". Only culprits that have been measured and thoroughly studied can be issued permits. We also store basic information on unknown transmitters under the "unknown" data object, as there is limited information known about them. The unknown transmitters are those whose source of transmission is unknown.

The transmitter's schema is likely to differ depending on the type of transmitter (i.e., intentional, culprit, or unknown). Therefore, the transmitter object should not have a fixed schema but should be allowed to vary with the transmitter type. Having a flexible schema is associated with greater scalability and performance as data is added to the database [181, 182]. Each sub-object inherits the transmitter's schema (parent object) and these are linked via an "ISA" relationship in inheritance. Inheritance in database design refers to a concept in which one database entity/data object can inherit properties and attributes from another entity [171]. From a specialization perspective in inheritance, a transmitter is an "intentional", a transmitter is a "culprit", or a transmitter is an "unknown transmitter". This is also regarded as a top-down approach to the relationship between the parent object and the sub-objects. As we move down the hierarchy, the specialization approach is based on grouping sub-objects with unique characteristics. This means that intentional, culprit, and unknown transmitter sub-objects have attributes and characteristics specific to themselves, on top of the parent's attributes.

We describe the modeling constraints set by astronomers and engineers at the MeerKAT radio telescope. We indicate the associations between data objects and specify the cardinality and participation using numerical or multiplicity notations.

Multiplicity notations in UML are used to represent cardinality and participation indicated as numerical notations (e.g., 0..1 – zero or one, 1..1 – one and only one, 1..M – one or many, or 0..M – zero or many). The first number shows the minimum number an object is involved in a relationship (participation), and the second number indicates the maximum number an instance of an object interacting with another (cardinality) [171]. For instance, a RECEIVER can capture one or more RFI EVENTs. Each EVENT may be received by one RECEIVER only. An RFI EVENT belongs to a specific TRANSMITTER of either type: intentional, culprit, or unknown. The relationship between the transmitter object and its sub-object is an "ISA" relationship. We represent the relationships between the transmitter and its sub-objects as *disjoint*. This is done to separate each transmitter by its source of transmission. For instance, the intentional transmitter's source can be legally known, the culprit transmitter's source can be determined by taking test measurements of a device, and the source of an unknown transmitter is not known at all. By enforcing that each sub-object belongs to only one specific transmitter, the degree of data integrity in the database can be guaranteed [183, 171]. Lastly, each TRANSMITTER may emit several RFI EVENTs. A PERMIT can be issued to each TRANSMITTER, although a transmitter may or may not have a PERMIT.

### 4.2.3 Logical Schema of the RFI database

We reduce the ERD in Figure 4.3 into a logical schema, showing each object along with its attributes and the relationships they have with attributes of other objects. We represent the schema by specifying the object name followed by the attributes in parentheses. Each object has an identifier or primary key (underlined attribute), which also serves as a reference inside other objects. These schemas include all the attributes of each data object required at the logical level of the RFI database.

```
receiver (receiverID, receiverName, nChannel, bitstream, nAccs, adcType, spectrumBits,
rfGain, bandwidth, lowFreq,  highFreq, rxLocation, rxDirection);

rfievent (eventID, receiverID, transmitterID, eventStartFreq, eventEndFreq, measDist, elevel, eirp,
MeerKATBand, polarization);

transmitter (transmitterID, txType, category, txDescription, txNotes,
bandname, bandcode, MeerKATBand, bandstartFreq, bandendFreq, bandNotes);

intentional (transmitterID, txType, category, txDescription, txNotes,
bandname, bandcode, MeerKATBand, bandstartFreq, bandendFreq, bandNotes,
status, startFreq, centFreq, endFreq, bandwidth, latitude, longitude, txLocation);

culprit (transmitterID, txType, category, txDescription, txNotes,
bandname, bandcode, MeerKATBand, bandstartFreq, bandendFreq, bandNotes,
manufacturer, model, serialNumber, testFacility, testDate, reportDate,
reportURL, reportID, reportComp, reportNotes, occupancy, testLatitude, testLongitude, testLocation);

unknown (transmitterID, txType, category, txDescription, txNotes,
bandname, bandcode, MeerKATBand, bandstartFreq, bandendFreq, bandNotes);

permit (permitID, transmitterID, rfiNotice, issueDate, expiryDate, usage, permitType,
deployDate, contactName, contactOrganisation,  contactEmail,
restrictionNo, restrictionLimited, restrictionUnlimited, rfiZoneMap,
useDayOrNight, useBeforeTime, useAfterTime, permitNotes);

catalog logical schema
engine (engineID, engineName, host, port, connectionProperties);

database (databaseID, databaseName, engineID);

object ( objectID, objectName, databaseID);
```

```
attribute (attributeNo, attributeName, datatype, description, objectID);
```

The receiver object stores the following information as attributes: receiver identification number, receiver name, number of channels, nature of the bit stream, number of accumulations per spectra, type of analog-to-digital converter, spectrum bits, receiver gain in decibels (dB), receiver bandwidth, lowest frequency, highest frequency in MHz, receiver location, and receiver direction. The receiver identification number ensures unique receiver records, and it cannot be null.

An RFI event object stores the event identification number, receiver identification number, transmitter identification number, start and end frequencies (in MHz) at which the event is detected, measurement distance to the antenna, effective radiated power of the RFI signal in dB, effective or equivalent isotropic radiated power of the RFI signal in decibel watt (dBW), MeerKAT frequency band in which the event is detected, and signal polarization. Polarization refers to the direction (either vertical or horizontal) in which an event is observed. The receiver ID inside the RFI event object references the receiver object, implying that the entries inside the RFI event object must match those in the receiver object. Similarly, the Transmitter ID inside the RFI event object references the transmitter object. Each RFI event record tracks a particular event, its associated transmitter, and the frequencies and direction at which it has been detected. This information is valuable for guiding radio astronomers in their observations.

The transmitter includes the transmitter identification number, transmitter type, transmitter category, transmitter description, notes or comments on the transmitter, the frequency band of the transmitter, code of the frequency band, MeerKAT band, start and end frequencies of MeerKAT band, and notes/comments on the MeerKAT frequency bands. However, the transmitter's schema can vary depending on the type of transmitter (i.e., intentional, culprit, or unknown). Recall that the relationship between the transmitter (parent) and the intentional, culprit, and unknown (sub-objects) is known as "ISA" or inheritance. In this case, the sub-objects inherit all properties and attributes of a parent object. Notice that the transmitter ID is a primary key in the transmitter, serving as both the primary key and foreign key in all the sub-objects. If a transmitter is intentional, it will include all the transmitter's attributes due to inheritance, along with its specific attributes, which include the operational status of a transmitter, start, central, and end frequencies of a transmitter, bandwidth of the transmitter, latitude, longitude, and location of the transmitter. These attributes are easily identifiable, as intentional transmitters are legally licensed and registered by ICASA to transmit in a specific location within a designated frequency range. Collecting these specific attributes in the database will help engineers in tracking and reporting any interfering transmitters.

If a transmitter is a culprit, its schema will include all attributes from the transmitter (parent) object, along with its specific attributes, such as manufacturer, model, serial number, test facility, test date, report date, report URL, report ID, company that drafted the report, report notes, polarization, latitude, longitude, and name of location where the test was done. It's important to note that the culprit object collects data on transmitters that have been detected, and their measurements are taken before taking the device to the site. Therefore, storing information about the manufacturer, model, and serial number provides specifications for the device on the site. Details about the test facility, date, and report indicate where the device was tested and when reports were completed.

If a transmitter is categorized as "unknown," it will inherit all of the attributes from the main transmitter object. However, for an unknown transmitter, there is

no specific information available about the transmitter's source, as it is unknown. Therefore, we assume a general or basic schema based on the parent object (transmitter). This is why the schema for an unknown transmitter is identical to that of the main transmitter object. Our schema is designed to be flexible and capable of accommodating any information that scientists may want to store about unknown transmitters. Additionally, attributes in the subobjects may appear redundant, but this method ensures that obtaining information about a transmitter does not require accessing two entities, i.e., the one corresponding to the subobject schema and the one corresponding to the parent schema. Also, no single record in the transmitter record is stored in more than one subobject, as this has been enforced with the disjoint constraints in the relationship

The permit schema consists of permit identification number, transmitter identification number, category of RFI permit, permit issue date, permit expiry date, indication of locations where permits are required, permit type, deployment date of the transmitter on-site, contact name and organization responsible for the transmitter, and email contact information. Additionally, we store the restriction identification number, limited restriction and unlimited restriction zones of a transmitter on the site, a map showing RFI zones, the times of the day or night a transmitter is allowed on the site, including the before and after times, and lastly, we collect notes/comments related to a specific permit.

Our catalog's logical schema includes four data objects: engine, database, object, and attribute, which represent the metadata. The engine object stores engine information, including the engine identification number, engine name, host machine, port number, and connection properties as attributes. The database data object stores information about databases, including the engine identification number and the database name. Meanwhile, the object data object stores the data object identification number, object name, and database identification number. The engine ID inside databases and the database ID inside objects act as references to the engine and the database, respectively. The attribute data object stores information about each attribute of every object, including attribute number, object identification number, attribute name, data type, and description. Each piece of information collected in the catalog serves as metadata for the RFI database. We do not represent the catalog schema in the ERD as it is not primary to the RFI database, despite playing a crucial role in providing all the necessary metadata.

### 4.2.4 Physical Design

This stage is dependent on both software and hardware. The goal is to create a physical model that can support diverse data sets, distinct data models, and multiple database systems without compromising database performance. Therefore, we have structured RFI data into three schemas (relational, columnar, and multidimensional arrays) that determine a particular access method. The relational schema is suitable for relational data, the columnar schema is best for key-value data, and the multidimensional schema is ideal for arrays. Consequently, we have created an RFI catalog schema to store all metadata using relational tables. We use predefined relation schema to ensure that the metadata schema remains consistent across multiple data stores in the cluster. Consistency is one of the properties offered by relational database systems [106]. Figure 4.4 shows a representation of the relation schema of the catalog database in a relational database system (PSQL).

The engine, database, object, and attribute relations define the RFI catalog, which is used to store all RFI metadata. A particular database is associated with at least one

```
CREATE TABLE IF NOT EXISTS rfi.engines (
       engineID serial PRIMARY KEY,
       engineName varchar(50) not null,
       host varchar(50) not null,
       port integer not null,
       connectionProperties varchar(100)
);

CREATE TABLE IF NOT EXISTS rfi.databases (
       databaseID serial PRIMARY KEY,
       databaseName varchar(50) not null,
       engineID serial REFERENCES rfi.engines (engineID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.objects (
       objectID serial PRIMARY KEY,
       objectName varchar(50) not null,
       fields text not null,
       databaseID serial REFERENCES rfi.databases(databaseID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.attributes (
        attributeNo serial PRIMARY KEY,
        attributeName varchar(50) not null,
        datatype varchar(50) not null,
        description text not null,
        objectID serial REFERENCES rfi.objects (objectID) ON DELETE CASCADE

);
```

FIGURE 4.4: Relational schema creation

database engine, but an engine can run multiple databases. This is why we create an engine identification ID ('engineID') inside the 'database' relation to associate a specific engine with multiple databases. We create the 'object' relation with a database identification ID ('databaseID') referencing the 'database' table. This implies that a specific RFI object belongs to one database, which is also associated with exactly one engine. However, a database can consist of several RFI objects. Lastly, we create an 'attribute' relation associated with the 'object' table. We include the object identification ID ('objectID') inside 'attribute' to indicate that a specific object has several attributes. It's important to note that the attribute identification number, object identification number, database identification number, and engine identification number serve as indexes in their respective tables. This allows for quick searches of specific attributes or fields; the RFI data objects they belong to; the databases containing those objects; and the storage engines running the databases within the cluster. This schema is well-suited for relational database systems such as Oracle, SQL Server, MySQL, and PostgreSQL.

We employ key-value pairs in column-oriented database systems to accommodate unstructured data because the schema is likely to change when we add data to the database. For example, the schema for 'unknown' transmitters consists of fewer fields compared to intentional and culprit transmitters due to the limited information available about unknown transmitting devices. Therefore, maintaining a flexible schema in a key-value fashion allows us to avoid unnecessary fields by not storing keys with null values. This provides an efficient means to reduce the number of null values, resulting in the efficient utilization of database memory and facilitating faster data access [113]. Key-value structures are

well-suited for data with no predefined structure; the key can easily be associated with a data value of any type, including text, logs, images/scans, video, and sensor data.

Figure 4.5 illustrates key-value pairs in the columnar schema of the column-oriented database system (Accumulo). Refer to Appendix H for full key-value schema for a single key (rfi001).

```
key   : columnFamily : columnQualifier : value
.....................:.............................
rfi001  transmitter  : transmitterID
rfi001  transmitter  : txType
rfi001  transmitter  : category
rfi001  transmitter  : txDescription
rfi001  transmitter  : txNotes
rfi001  transmitter  : status
rfi001  transmitter  : startFreq
rfi001  transmitter  : centFreq
rfi001  transmitter  : endFreq
rfi001  transmitter  : bandwidth
rfi001  transmitter  : latitude
rfi001  transmitter  : longitude
rfi001  transmitter  : txLocation
rfi001  transmitter  : manufacturer
rfi001  transmitter  : model
rfi001  transmitter  : serialNumber
rfi001  transmitter  : testFacility
rfi001  transmitter  : testDate
rfi001  transmitter  : reportDate
rfi001  transmitter  : reportURL
rfi001  transmitter  : reportID
rfi001  transmitter  : reportComp
rfi001  transmitter  : reportNotes
rfi001  transmitter  : occupancy
rfi001  transmitter  : bandCode
rfi001  transmitter  : bandname
rfi001  transmitter  : MeerKATBand
rfi001  transmitter  : bandstartFreq
rfi001  transmitter  : bandendFreq
rfi001  transmitter  : bandNotes
rfi001  receiver   : receiverID
rfi001  receiver   : receiverName
```

FIGURE 4.5: A sample Accumulo's key-value schema

The columnar schema includes transmitter, receiver, and permit data objects or 'columnFamily'. A column family is a group of columns that are stored together as a unit, preferably in one location, whereas a column qualifier is a field associated with storing data within a specific column family. Each columnFamily stores data in a key-value fashion, where the key ('RFI identification number' — e.g., rfi001) is processed chronologically as the first RFI entry. The value can consist of data of any type, such as strings (e.g., 'Electric fence at Klipkolk'), issue date ('01/08/2010'), and coordinates ('-30.438867, 21.120959'). Retrieving RFI data is easy because we associate a key with data in the form of text, RFI reports, and RFI scans/images.

The transmitter object, which includes intentional, culprit, and unknown sub-objects, is implemented as embedded data objects using key-value pairs in JSON documents (see Appendix G). Embedded data objects provide a simple view of multiple data types and speed up search results due to a single index associated

with that object, as opposed to many indexes from different objects. The use of multiple indexes incurs a storage overhead that might compromise the speed of searches across multiple storage systems [163]. Additionally, this approach prevents the duplication of fields and ensures proper memory utilization by storing only the fields or 'columnQualifiers' required for that specific transmitter. The receiver and permit objects are likely to have the same fields from one record to another, but this is not the case for the transmitter object.

Key-value pairs are optimized for faster data access when searching, utilizing both key and column information concurrently. For instance, when we specify a key (e.g., rfi001) and a transmitter column, rfi001 points to the exact data location of the object (transmitter) instead of scanning all data objects in the cluster. Key-value data is stored using column-oriented database systems such as Accumulo, known for its good performance in cluster environments [117].

We presented the sequential array data in a tabular fashion similar to a typical relational schema. In a multi-dimensional array model, data can be indexed and accessed separately in each dimension (e.g., frequency and time). On the other hand, in the relational model, data is accessible and presented as one whole unit (table). Figure 4.6 represents a schematic representation of an array-oriented model. This is implemented in an array database system (such as SciDB).

```
CREATE ARRAY RFIevent <
    eventID integer
    transmitterID integer
    receiverID integer
    eventStartFreq double
    eventEndFreq double
    elevel double
    measDist double
    eirp double
    MeerKATBand string
    polarisation string
>[i=0:*,1000000,0];
```

FIGURE 4.6: A sample SciDB's array schema.

We created an array object called 'RFIevent' to store data from RFI events taken from several measurements of different transmitters. A single transmitter can contain multiple events, with the total number of events potentially being quite large. For each event, we record eventID, receiverID, transmitterID, eventStartFreq, eventEndFreq, elevel, measDist, MeerKATBand, and polarization. The 'transmitterID' and 'receiverID' fields inside the event object are used to trace which transmitter, an event belongs to and on which receiver it was captured. Further, we store the start and end frequencies of an event, the signal strength of an event (elevel in dB), measurement distance from the antenna in meters, eirp power in dBW, the MeerKATBand, and the direction of the event.

Each event behaves differently, and to understand the behavior of each event, we record each parameter. For instance, consider the 'Two-way radio system,' which can emit two events. Event 1, identified as 'rfi001,' belongs to transmitter 'RFI1402-0620,' was detected at receiver 'rec001,' and spans the frequency range from 120 MHz to 134 MHz. It emits a signal power of 70 dB, measured at a distance of 10 meters from the antenna, has an eirp power of -126 dBW, and operates in the MeerKAT band known as 'Hera' with vertical polarization. Event 2, also from

'RFI1402-0620' and detected by 'rec001,' starts at 240 MHz and ends at 270 MHz. It emits a signal power of 30 dB, was measured at a distance of 10 meters from the antenna, has an eirp power of -131 dBW, and operates in a non-MeerKAR band with horizontal polarization. Notice that these two events are from the same transmitter. Imagine a high number of transmitters emitting multiple events; this scenario is likely to generate a large data set that can be efficiently compressed into an array structure.

An array schema allows us to set up an array data store for the extensive RFI events data spanning large measurements taken across a wide frequency spectrum. Such an array schema can accommodate a maximum of one million entries (i = 1000000). One advantage of array stores is their ability to facilitate data compression into smaller chunks, thereby minimizing disk space usage and enabling quick access [184, 147]. We implemented the array model using the SciDB DBMS. In addition to handling a large number of entries, SciDB provides scientists with a broad range of analytics tools for exploring various phenomena within their multi-dimensional data sets.

Due to the multiple schemas of RFI data, we adopted the Polystore framework (described in Section 3.4), which spans various data models, languages, and storage engines. We use three categories for data stores: PostgreSQL, Accumulo, and SciDB (See Figure 4.7).

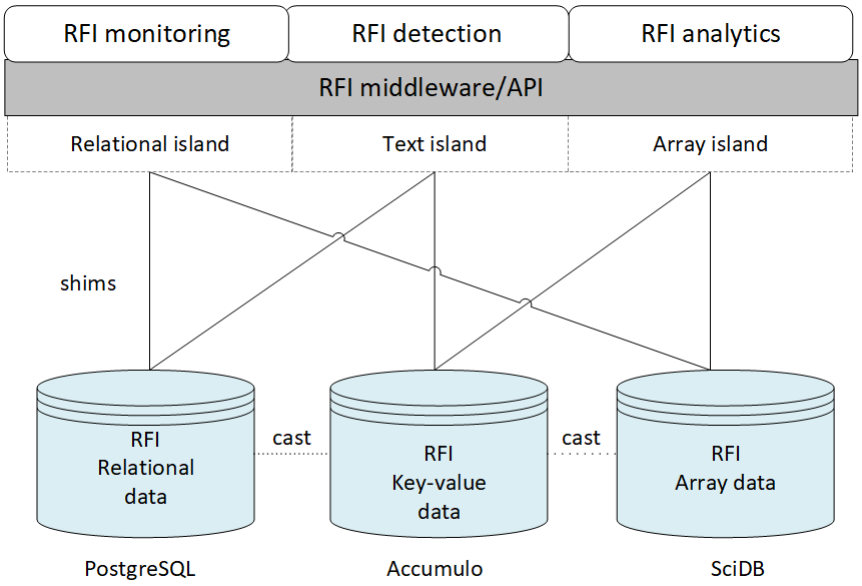



FIGURE 4.7: A polystore design of the RFI database.

PostgreSQL stores the RFI catalog using a relational schema, which includes engines, databases, objects, and attributes as metadata. Accumulo stores key-value data (text, images, documents, and reports) using a columnar schema and holds transmitter (intentional, culprit, and unknown), receiver, and permit data objects. We store array data in SciDB using a multidimensional array schema, which houses the RFI event data object. Each schema is well-suited to its respective data store. We argue that storing RFI data objects in their appropriate data stores while maintaining their relationships ensures the proper translation of the logical design into the physical design. Figure 4.7 shows the polystore implementation design suitable to store RFI data.

Lastly, the polystore design enables engineers to load data through RFI monitoring and detection applications, while scientists can conduct quick searches and detailed analyses using an integrated API. We discuss the detailed implementation of each store component in the next chapter.

### 4.2.5 Alternative Design

We present an alternative design to evaluate the performance of the polystore database system (BIGDAWG) compared to storing all data in PostgreSQL, particularly since PostgreSQL performed best in the experimental evaluation. We present an alternative design based on the conceptual design. The goal is to break down data objects with several fields or attributes into one or more manageable data objects. The main object is the parent, and the sub-objects become the child entities. We model an identifying relationship between the parent object and the child object. This means an instance in the child object is identified through an association with the parent entity. That's why the primary key of the parent forms part of the primary key in the child entity. With this kind of association, we ensure that the same level of integrity is enforced as when the object was treated as a collective unit. Figure 4.8 is an alternative logical ERD of the RFI database.

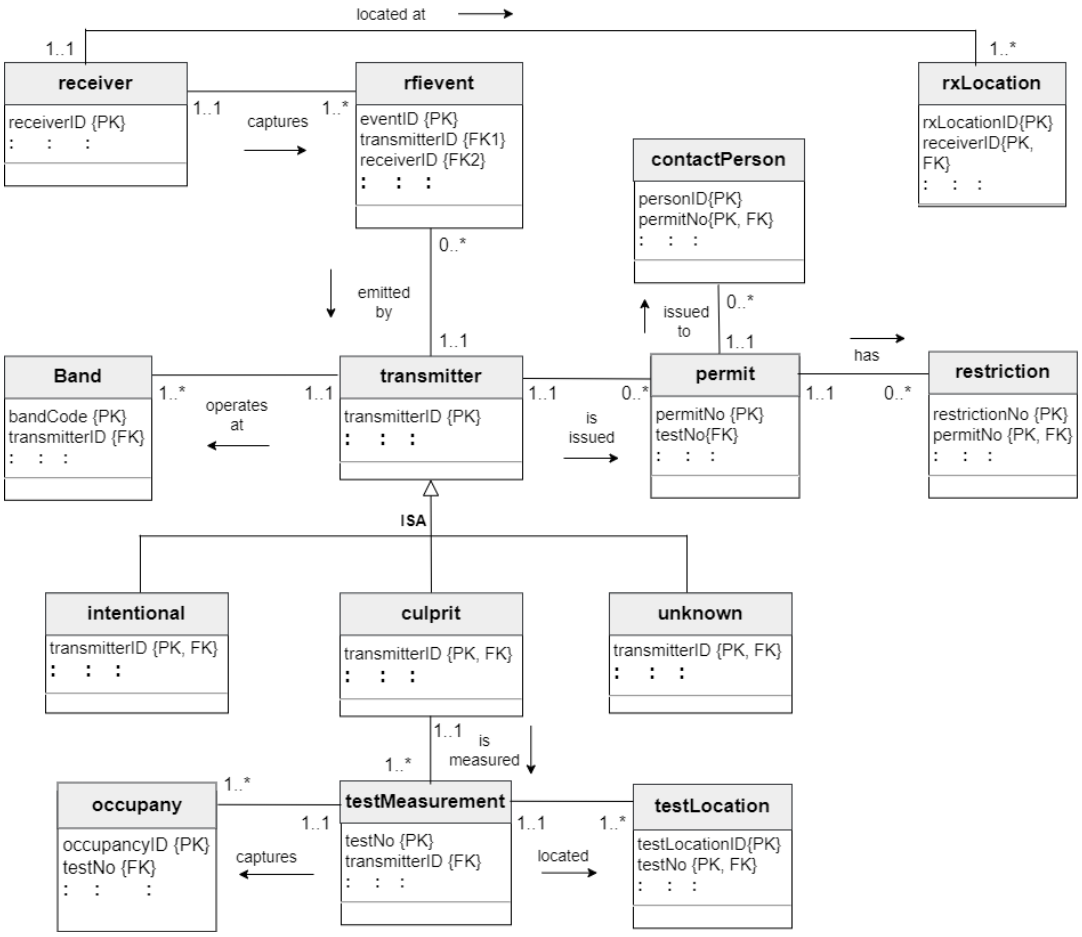



FIGURE 4.8: Alternative ERD of the RFI database (Refer to the logical schema showing all attributes in each data object (Section 4.2.6)

Notice that Figure 4.8 in the alternative design consists of more data objects compared to Figure 4.3. The ERD of the alternative design includes receiver, receiver

location (rxLocation), rfievent, transmitter, intentional, culprit, unknown, band, testMeasurement, testLocation, permit, contactPerson, and restriction. In contrast, the ERD in Figure 4.3 includes receiver, rfievent, transmitter, intentional, culprit, unknown, and permit data objects.

The receiver object in Figure 4.3 forms two relations in the alternative design: receiver and receiver location. The receiver object includes receiverID, receiver name, number of channels, nature of the bit stream, number of accumulations per spectra, type of analog-to-digital converter, spectrum bits, receiver gain, receiver bandwidth, lowest frequency, and highest frequency, whereas receiver location consists of the locationID of the receiver, receiverID, and the name of the location of the receiver as attributes. The receiverID inside the receiver location object references the receiver object. The cardinality between receiver and receiver location is one to many, which suggests that a receiver can be placed in several locations during RFI detection on the site, while a specific location can have only one receiver.

An RFI event object in the alternative design stores the event identification number, receiver identification number, transmitter identification number, start and end frequencies (in MHz) at which the event is detected, measurement distance to the antenna, effective radiated power of the RFI signal (elevel)in dB, effective or equivalent isotropic radiated power (eirp) of the RFI signal in dBW, MeerKAT frequency band in which the event is detected, and signal polarization. Polarization refers to the direction (either vertical or horizontal) in which an event is observed. The receiver ID inside the RFI event object references the receiver object, implying that the entries inside the RFI event object must match those in the receiver object. Similarly, the Transmitter ID inside the RFI event object references the transmitter object. A record in the RFI event object can track a particular event, its associated transmitter, and the frequencies and direction at which it has been detected. This information is valuable by providing quick guidance to the radio astronomers during their observations. It's worth noting that the RFI event schema in the alternative design is similar to the first design (Figure 4.3) and stores all fields in one relation. The RFI event in both designs consists of only a few attributes that can be easily handled by a single relation

The transmitter object maintains an identifying relationship with the band object and an 'ISA' relationship (inheritance) with the intentional, culprit, and unknown sub-objects. Not only do the sub-objects inherit the transmitter's attributes, but they also inherit their underlying relationship with the band object. The transmitter object will store attributes such as transmitterID, type of transmitter (txType), category, txDescription (description of transmitter), status, startFreq, centFreq, endFreq, and bandwidth. Notice that the transmitter object in the alternative design consists of fewer attributes than in Figure 4.3.

The transmitter's band information is stored in a separate relation known as 'band'. This includes the code and name of the band, transmitterID, MeerKATband, start and end frequency of each band, as well as comments or notes for a particular band. The transmitterID inside Band objects forms part of the band's composite primary key (transmitterID and bandCode).

Intentional, culprit, and unknown transmitters each have specific attributes in their respective schemas, along with common attributes of the transmitter. Intentional transmitter-specific attributes include the transmitter's operational status, start, central, and end frequencies of a transmitter, bandwidth of a transmitter, latitude, longitude, and the name of the transmitter's location (see logical schema in Section 4.2.6). The rest of the attributes are common attributes of the transmitter, which is why they are identical. The specific attributes of

an intentional transmitter are easily accessible, as they are legally registered and well-known transmitters (e.g., FM radio). However, this is not the case with the unknown transmitter object, which includes common attributes of the transmitter because there is limited information available about them. Hence, they have a basic schema for the transmitter.

The culprit object includes all transmitter attributes, including its specific attributes i.e., manufacturer, model, and serial number of a transmitter. In the alternative design, the culprit object differs from that in Figure 4.3 in such a way that it forms two additional relations with its underlying attributes. These additional relations include test measurement (testMeasurement) and test location (testLocation). We store test measurements and test locations for each culprit separately, as there are likely to be several measurements taken from various locations associated with a single device or culprit.

Furthermore, a culprit object establishes a one-to-many relationship with the testMeasurement object, which in turn forms a one-to-many relationship while also creating an identifying relationship with the testLocation data object. In other words, a single test measurement for a specific culprit can be conducted at several locations. The existence of the testLocation object depends on the existence of the testMeasurement object through a one-to-many relationship, as indicated by the inclusion of 'testNo' as part of the foreign key in the testLocation object. Lastly, the testMeasurement stores the test number, transmitterID, test facility, test date, report date, report URL, reportID, a company that drafts the report, and notes or comments on a specific report, whereas the testLocation stores testLocationID, test number, name, longitude, and latitude of the test location, as shown in Section 4.2.6.

The permit object in the alternative design forms two sub-objects: contactPerson and restriction. Both objects exist due to the existence of the permit object. Without a permit object, we cannot have these sub-objects; hence, we indicate optional participation between the permit and its sub-objects. A permit may have multiple contact persons and can suggest multiple restrictions at the same time. We store attributes such as personID, permitID, name, organization, and email for the contact person data object. Permit restriction details, including restriction number, restricted and unrestricted zones, maps showing RFI restricted zones, and permitted times of the day or night for device access to the site, are all stored within the restriction data object. Lastly, we maintain the same modeling and relationship constraints observed by the main objects and all their sub-objects, as when they were a collective unit.

### 4.2.6 Logical Schema for alternative design

We reduce the alternative ERD to logical schema showing more data objects taken into consideration. We list the name of the data object followed by its attributes enclosed in parentheses. We indicate the primary key as underlined and also include the foreign keys where necessary. These schemas show individual data objects shown in Figure 4.8 and all their attributes.

```
receiver (receiverID, receiverName, nChannel, bitstream, nAccs, adcType,
spectrumBits, rfGain, bandwidth, lowFreq, highFreq);

rxLocation (rxLocationID, receiverID, rxLocationName, rxDirection);


rfievent (eventID, receiverID, transmitterID, eventStartFreq, eventEndFreq, elevel, measDist,
eirp, MeerKATBand, polarization);

transmitter (transmitterID, txType, category, txDescription, txNotes);
```

```
band (bandCode, transmitterID, bandname, MeerKATBand, bandstartFreq,
bandendFreq, bandNotes)

intentional (transmitterID, txType, category, txDescription, txNotes,
status, startFreq, centFreq, endFreq, bandwidth, latitude, longitude, txLocation);

culprit (transmitterID, txType, category, txDescription, txNotes,
manufacturer, model, serialNumber);


testMeasurement (transmitterID, testFacility, testDate, reportDate,
reportURL, reportID, reportComp, reportNotes);

occupancy (occupancyID, testNo, month, nights, evenings, strong,
moderate, weak);

testLocation (testLocationID, testNo, testLocationName, testLongitude,
testLatitude)

unknown (transmitterID, txType, category, txDescription, txNotes);

permit (permitID, transmitterID, rfiNotice, issueDate, expiryDate, usage,
permitType, deployDate, permitNotes);

contactPerson(permitID, personID, contactName, contactOrganisation,
contactEmail);

restriction (restrictionNo, permitID, restrictionLimited,
restrictionUnlimited, rfiZoneMap, useDayOrNight, useBeforeTime, useAfterTime);


catalog logical schema
engine (engineID, engineName, host, port, connectionProperties);

database (databaseID, databaseName, engineID);

object ( objectID, objectName, databaseID);

attribute (attributeNo, attributeNo, attributeName, datatype, description, objectID);
```

The alternative design is created assuming a single modality, i.e., the relational model. As a result, many data objects and relationships have been established to accommodate the original data stored across three systems (PSQL, Accumulo, and SciDB). We model the design by transforming Accumulo's key-value data and SciDB's array data into relational data in PSQL without compromising the data's integrity. Refer to Appendix F for a detailed description of each attribute and its corresponding data objects. Lastly, we discuss the implementation of the alternative design in the next chapter.

# Chapter 5

# Implementation and Evaluation

Here, we discuss the implementation and evaluation of the physical database models. First, we describe the implementation of the RFI storage according to the polystore architecture. The RFI storage consists of three databases: the RFI catalog to stores all RFI metadata using relational tables; the RFI array database responsible for RFI sequential data, such as frequency and time data, using multidimensional arrays; and the RFI measurement data structured as key-value pairs. Furthermore, we implement an alternative model using a single data model where all data is stored in one data store under the polystore framework. Thereafter, we explain the experimental design and the experiments in different test environments to emulate our RFI monitoring use case. We test the database models with increasing data volumes, multiple users, varying workloads, bulk uploads, and API environments. The acceptable latency without interruption is one second or less [168, 157, 185]. A latency of less than 1 second is primarily attributed to the requirements of many real-time systems, in line with industry practices for achieving acceptable overall system performance. Over the years, industry practitioners have made efforts to reduce latency in various systems through technological advancements and user experience, with 1 second being a commonly acceptable threshold [186, 187]. However, it's important to note that different systems may have varying latency thresholds depending on their specific needs. For instance, in real-time applications (high-frequency communications or interactive) where immediate responsiveness is crucial, latency thresholds are often much lower than 1 second to ensure accurate processing.

Previous work on new and scalable databases have not decided on the standard acceptable values for upload speed, download speed, and response time [102, 160, 162].

## 5.1 Storage

We used Docker [188] to emulate an environment in which RFI data is processed using several applications (such as RFI monitoring, detection, and complex analytics) and is stored in different Stores. Docker is a container-based virtualization technology that allows isolated applications to be created, deployed, and executed. This technology enables the creation of a system that houses several Docker containers working in the same cluster. These containers are isolated from each other but can communicate through well-defined channels [189]. Docker shares a single OS kernel with several containers that have less boot time. Other benefits of Docker include: (a) it is portable – which means an application and all its dependencies are bundled into a single container that is independent of the host OS kernel version, platform distribution or deployment model - and (b) lightweight – which implies

that docker requires minimal resources, as its images are small [190]. These benefits facilitate rapid delivery and reduce the time needed to deploy an application. Docker is the most efficient environment for running multiple applications that suffer from integration and interoperability complexities [191], thus suitable for this work.

### 5.1.1 RFI Database Cluster

We used Docker containers to deploy the database cluster using Docker version *18.09.3, build 774a1f4*. We ran the Docker image provided in the reference implementation (BigDAWG) [130] to set up the database cluster on the host. Our host machine runs the Ubuntu Operating system version *Ubuntu 20.04.3*, *25 GB* of RAM and *11 TB* of disk capacity. The cluster consists of several containers of three databases: PSQL, SciDB, and Accumulo. Data access can occur within or between containers. While a polystore can have several containers, we set up only data stores that are suitable for storing RFI data.

TABLE 5.1: Showing database engines

| **engine ID** | **name** | **host** | **port** | **connection properties** |
|---|---|---|---|---|
| 0 | postgres0 | bigdawg-postgres-catalog | 5400 | PostgreSQL 9.4.5 |
| 1 | postgres1 | bigdawg-postgres-data1 | 5401 | PostgreSQL 9.4.5 |
| 2 | postgres2 | bigdawg-postgres-data2 | 5402 | PostgreSQL 9.4.5 |
| 3 | scidb_local | bigdawg-scidb-data | 1239 | SciDB 14.12 |
| 4 | saw Zookeeper | zookeeper.docker.local | 2181 | Accumulo 1.6 |

Table 5.1 lists the database engines, hosting, and connection information. This detail is managed and well-coordinated across multiple containers using the polystore middleware. The Docker cluster consists of five database engines of three types: PostgreSQL, SciDB, and Accumulo. Three PostgresSQL engines are used to store different RFI data and to enable faster connections to data access than SciDB and Zookeeper(Accumulo), whose connections depend on running several applications within the cluster before data is accessed [121]. Each database engine has a unique name and identifier (i.e., *engineID = 0*), which are used during data placement from one engine to another [192]. Each engine is hosted on a separate host with a unique hostname (i.e., bigdawg-postgres-catalog). Using Docker containers with the *docker create container* command, we created several hosts. Each container was allocated a port number that was published within the cluster so that other containers could connect and communicate data. Postgres0, postgres1, and postgres2 use the same type of connection (i.e., PostgreSQL 9.4.5) because they use the same type of engine (PostgreSQL). *SciDB_local* uses SciDB 14.12 connections in the SciDB environment, whereas Zookeeper [193] uses Accumulo 1.6 connections. ZooKeeper is an application within the SciDB cluster responsible for coordinating and synchronizing services in an ecosystem with distributed applications where coordination is a core requirement. The connections enable the clients and the cluster host to communicate with different containers. This information is crucial when data is moved from one engine to another [194]; for instance, a query to return catalog data and measurement key-value data from *postgres0* and *SciDB_local*, respectively, to a user or client. Such a query requires engine name, port, and connection details to complete a transaction.

### 5.1.2 RFI Data Stores

We created six databases in containers supported by the different database engines (Table 5.1). The database table (Table 5.2) references the engine information via *engineID* as a foreign key. The relationship between the engine and database tables is one-to-many: a database engine is shared by at least one database. However, a database can belong to a single database engine only. Table 5.2 lists each database with its corresponding engines. The information consists of the *databaseID* as a unique identifier, engine identifier (*engineID*), and the name of the database.

TABLE 5.2: Showing each database created on a suitable engine

| database ID | engine ID | name |
|---|---|---|
| 0 | 0 | RFI_catalog_data |
| 1 | 0 | RFI_schemas_data |
| 2 | 1 | RFI_postgres_data |
| 3 | 2 | RFI_dataset |
| 4 | 3 | RFI_scidb_data |
| 5 | 4 | RFI_accumulo_data |

We have six databases and five engines, each identified by databaseID and engineID, respectively. The RFI_catalog_data database is identified with databaseID = 0 and engineID = 0 (Table 5.2). RFI_catalog_data is a database that contains metadata, storing definitions of database objects such as tables, engines, systems, and cluster information. The RFI_catalog_data stores small records that fit into relational tables found in the PostgreSQL engine.

The RFI_schemas_data database is identified with databaseID = 1 and engineID = 0 (Table 5.2). The RFI_schemas_data is a database that contains schematic information for all data, including fields and the associated database. Similarly, RFI_schemas_data stores small records that fit into relational tables found in PostgreSQL. Therefore, RFI_catalog_data and RFI_schemas_data run on the same storage engine.

The RFI_postgres_data database stores analytical and time-series data, such as RFI occupancy data, including daily and weekly RFI occupancy statistics. This data consists of numerical values with a few fields that fit into the relational tables of PostgreSQL. RFI signal levels and time duration data on affected frequency channels are stored in relational tables and computed daily, weekly, or monthly to establish the RFI occupancy at the site. The database and engine are unique, with databaseID = 2 and engineID = 1 (Table 5.2). This database is located in a separate container from the RFI_catolog_data and the RFI_schemas_data to avoid performance overheads.

The RFI_dataset database, identified by databaseID = 3 and engineID = 2 (Table 5.2), is supported by Postgres. This is a database container reserved for bulk loading RFI data into respective databases. It consists of data files in HDF5, CSV, and JSON yet to be processed and loaded into different databases in the cluster. It serves as the source database where loading scripts are prepared to transfer data to other databases in different locations within the cluster (i.e., RFI_postgres_data, scidb_data, and RFI_accumulo_data). Similarly, we isolate it from other databases to ensure that bulk loading and processing do not interfere with other database operations. This approach significantly minimizes performance overheads associated with a cluster running several applications concurrently.

The array database (*RFI_scidb_data*) identified by *databaseID = 4* and *engineID = 3* stores all RFI array data objects. Sequential data of RFI events, such as start and

end frequency, measurement distance, signal strength or effective radiated power, equivalent isotropically radiated power, MeerKATband, and polarization, are stored in the RFI event array database. The database tracks the source of each RFI event by transmitter identifier (transmitterID). A transmitter emits several RFI events at different times and frequencies. Therefore, storing such data requires an array database that facilitates the storage of several entries of multiple dimensions.

Lastly, the *RFI_accumulo_data* stores key-value RFI data of many types belonging to transmitters, receivers, and permits including transmitter's measurement data, notes/comments from scientists, permit details such as expiry times, and location details such as geographical coordinates. The database is identified by *databaseID = 5* and *engineID= 4* (Table 5.2). The database is suitable for storing a record with numerous fields a single unit representing a key and associated value, as opposed to storing several columns in one table.

Table 5.3 presents information about all data objects in the RFI database. This can be used to trace a specific data object, identify all its fields, and determine its primary database. Each data object is identified by a unique identifier (objectID) and its associated database. For example, the (*rfi.transmitter*) object includes the fields that a transmitter holds. This information is crucial for queries fetching numerous fields across different objects. Therefore, specifying the objectID will instruct the query to index all associated databases for that specific data object.

TABLE 5.3: Showing each data object created in a specific database location.

| **object id** | **objectName** | **fields** | **database id** |
|---|---|---|---|
| 1 | rfi.attribute | attributeNo, attribute, dataType, description, objectID | 0 |
| 2 | rfi.engine | engineID, engineName,host,port, connectionProperties | 0 |
| 3 | rfi.database | databaseID, databaseName, engineID | 0 |
| 4 | rfi.object | objectID, objectName, databaseID | 0 |
| 5 | rfi.receiver | receiverID, receiverName, nChannel, bitstream, nAccs, adcType, spectrumBits, rfGain, bandwidth, lowFreq, highFreq, location, direction | 5 |
| 6 | rfi.transmitter | transmitterID, txType, category, txDescription, txNotes bandcode, bandName MeerKATBand, bandstartFreq, bandendFreq, bandNotes, status, startFreq, centFreq, endFreq, bandwidth, latidute, longitude, txlocation manufacturer, model, serialNumber, testFacility, testDate, reportDate, reportURL, reportID, reportComp, reportNotes, testLatitude, testLongitude, testLocation | 5 |
| 7 | rfi.permit | permitID, transmitterID rfiNotice, issueDate, expiryDate, usage, permitType, deployDate, contactName, contactOrganisation, contactEmail, restrictionNo, restrictionLimited, rfiZoneMap, restrictionUnlimited, useDayOrNight, useBeforeTime, useAfterTime, permitNotes | 5 |
| 8 | rfi.event | eventID, receiverID, transmitterID, eventStartFreq, eventEndFreq, measDist, elevel, eirp, MeerKATBand, polarization | 4 |
| 9 | rfi.occupancy | occupancyID, testNo, month, nights, evenings, strong, moderate, weak | 2 |
| 10 | rfi.schemas | objectID, objectName, fields, databaseID | 1 |

Table 5.3 shows ten main objects created in the RFI database, including attribute, engine, database, object, receiver, transmitter, permit, event, occupancy, and schemas. Each has a unique identifier and a corresponding database. The table indicates that each data object can be associated with exactly one database, but

a database can contain multiple data objects. This is illustrated by attributes, engine, database, and objects associated with the same database, RFI_catalog_data (databaseID = 0). These objects consist of a few fields that fit well in relational tables.

The receiver, transmitter, and permit data objects are created in RFI_accumulo_data (databaseID = 5). Notice that these objects consist of numerous fields that do not fit well in a relational table format. A relational table with so many columns is considered a large table, which affects data retrieval due to the longer time taken to scan through all columns. For example, the transmitter consists of embedded attributes belonging to its sub-objects: intentional, culprit, and unknown transmitters. The RFI event data object is created in the RFI_scidb_data database, identified by databaseID = 4. The events object consists of fields storing a sequence of array RFI data measured in time and frequency dimensions. The RFI occupancy and schemas data object consist of a few fields that fit well in relational tables. The occupancy object associates with the RFI_postgres_data database identified by databaseID = 2, whereas schemas are created in RFI_schemas_data (databaseID = 1). Refer to Table 5.4 for a detailed description of each field and its corresponding object.

Table 5.4 shows all attributes, data types, descriptions, and their associated objects. Each attribute in the RFI database is tracked by the attribute number, indicating the object and store location within the cluster. For example, attributeNo, attributeName, dataType, and description all belong to the attribute object (Table 5.3), for which the object can be further traced to the database (Table 5.2) it belongs, and up to the engine (Table 5.1) running a specific database. In addition, the association between attribute and object entities is indicated in such a way that one object can have many attributes; however, an attribute belongs to one and only one object. This is why attributeNo, attributeName, dataType, and description are associated with the same object, rfi.attribute (objectID = 1).

TABLE 5.4: Showing attributes and their associated data objects (refer to Appendix F for full list of attributes)

| attribute no. | attribute | data type | description | Object id |
|---|---|---|---|---|
| 1 | attributeNo | integer | attribute catalog identification number | 1 |
| 2 | attributeName | varchar(50) | name of the attribute in a catalog | 1 |
| 3 | dataType | varchar(50) | attribute data type | 1 |
| 4 | description | text | attribute description | 1 |
| 5 | engineID | integer | db engine identification number | 2 |
| 6 | engineName | varchar(50) | db engine name | 2 |
| 7 | host | varchar(50) | hostname of the database | 2 |
| 8 | port | integer | port number at the host machine | 2 |
| 9 | connectionProperties | varchar(100) | connection details (engine version) | 2 |
| 10 | databaseID | integer | database identification number | 3 |
| 11 | databaseName | varchar(50) | database name | 3 |
| 12 | objectID | integer | data object identification number | 4 |
| 13 | objectName | varchar(50) | data object name | 4 |
| 14 | receiverID | varchar(25) | receiver identification number | 5 |
| 15 | receiverName | varchar(50) | receiver name | 5 |
| 16 | nChannel | integer | designated number of channels | 5 |
| 17 | bitstream | varchar(50) | nature of bits streams | 5 |
| 18 | nAccs | integer | Number of accumulations per spectra | 5 |
| 19 | adcType | varchar(50) | type of analog-to-digital converter (adc) | 5 |
| 20 | spectrumBits | double | number of spectrum bits per channel | 5 |
| 21 | rfGain | double | receiver antenna gain | 5 |
| 22 | bandwidth | double | receiver bandwidth (applies to transmitters) | 5 |
| 23 | lowFreq | double | receiver lowest frequency spectrum | 5 |
| 24 | highFreq | double | receiver highest frequency spectrum | 5 |
| 25 | rxlocation | varchar(50) | name of receiver location | 5 |
| 26 | rxdirection | varchar(50) | receiver direction | 5 |
| 27 | transmitterID | varchar(25) | transmitter identification number | 6 |
| 28 | txType | varchar(50) | type of transmitter | 6 |
| 29 | category | text | short transmitter description | 6 |
| 30 | txDescription | text | detail transmitter description | 6 |
| 31 | txNotes | text | comments or notes on a transmitter | 6 |
| 32 | status | varchar(50) | status of operation of a transmitter | 6 |
| 33 | startFreq | double | transmitter start spectrum frequency | 6 |
| 34 | centFreq | double | central spectrum frequency | 6 |
| 35 | endFreq | double | transmitter end frequency | 6 |
| 36 | bandcode | Varchar(50) | band code of a transmitter | 6 |
| 37 | bandname | varchar(50) | band name of a transmitter | 6 |

Figure 5.1 shows the relational schemas used to create the structure of a relational database that includes sample data illustrated in Tables 5.1, 5.2, 5.3, and 5.4, being implemented in PSQL DBMS.

```
CREATE TABLE IF NOT EXISTS rfi.engines (
       engineID serial PRIMARY KEY,
       engineName varchar(50) not null,
       host varchar(50) not null,
       port integer not null,
       connectionProperties varchar(100)
);

CREATE TABLE IF NOT EXISTS rfi.databases (
       databaseID serial PRIMARY KEY,
       databaseName varchar(50) not null,
       engineID serial REFERENCES rfi.engines (engineID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.objects (
       objectID serial PRIMARY KEY,
       objectName varchar(50) not null,
       fields text not null,
       databaseID serial REFERENCES rfi.databases(databaseID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.attributes (
        attributeNo serial PRIMARY KEY,
        attributeName varchar(50) not null,
        datatype varchar(50) not null,
        description text not null,
        objectID serial REFERENCES rfi.objects (objectID) ON DELETE CASCADE

);
```

FIGURE 5.1: Relational schema creation

The receiver, transmitter, and permit data are examples of key-value data objects created using the Accumulo database system. These data objects are logically linked by the RFI identification number, indexed as 'rfi001,' which returns all key-value data from each object associated with this particular key. Figure 5.2 illustrates the schema creation of the key-value database. It consists of several inserts specifying a unique key with all its associated data objects (columnFamily) along with their attributes/fields (columnQualifier) and values. The key-value schema includes sample data (receiver, transmitter, permit) associated with a single key, rfi001, being implemented in Accumulo DBMS.

```
        key   columnFamily : columnQualifier value
.....................................................
insert rfi001  transmitter  : reportID M2901-0000-001
insert rfi001  transmitter  : reportComp MESA
insert rfi001  transmitter  : reportNotes unknown
insert rfi001  transmitter  : occupancy unknown
insert rfi001  transmitter  : bandCode "B Cast"
insert rfi001  transmitter  : bandname "Broadcasting (FM radio)"
insert rfi001  transmitter  : MeerKATBand Hera
insert rfi001  transmitter  : bandstartFreq 87.5
insert rfi001  transmitter  : bandendFreq 108
insert rfi001  transmitter  : bandNotes "fm broadcasts happen in this band"
insert rfi001  receiver  : receiverID recv001
insert rfi001  receiver  : receiverName RTA
insert rfi001  receiver  : nChannel 1024
insert rfi001  receiver  : bitstream unknown
insert rfi001  receiver  : nAccs 10.50
insert rfi001  receiver  : adcType unknown
insert rfi001  receiver  : spectrumBits 100
insert rfi001  receiver  : rfGain 60
insert rfi001  receiver  : bandwidth 200
insert rfi001  receiver  : lowFreq 180
insert rfi001  receiver  : highFreq 2300
insert rfi001  receiver  : rxLocation Karoo
insert rfi001  receiver  : rxDirection Horizontal
insert rfi001  permit  : permitID RFI1807-0014-001
insert rfi001  permit  : rfiNotice "Type A"
insert rfi001  permit  : issueDate 01/01/2010
insert rfi001  permit  : expiryDate 01/06/2011
```

FIGURE 5.2: Key-value schema creation. Refer to Appendix H to see the insertion of all key-value schema associated with a single key (rfi001)

The RFI event is an example of an array data object created using the SciDB database system. The data consists of a sequence of events measured at specific frequencies. A single data object is capable of storing over a million entries or records. This is one of the benefits of arrays over relational tables [22]. Figure 5.3 shows the array schema used to create the structure of an array database that includes sample data (rfievent) illustrated in Figure 5.4, being implemented in SciDB DBMS.

```
CREATE ARRAY RFIevent <
    eventID integer
    transmitterID integer
    receiverID integer
    eventStartFreq double
    eventEndFreq double
    elevel double
    measDist double
    eirp double
    MeerKATBand string
    polarisation string
>[i=0:*,1000000,0];
```

FIGURE 5.3: Array schema creation.

Figure 5.4 illustrates a sample array of data for the RFI event object. The first four events belong to the same transmitter, captured on a specific receiver (recv001),

measured over a range of frequencies. If we store such data for hundreds of transmitters, there is likely to be a high number of event entries.

```
eventID,transmitterID,receiverID,eventStartFreq,eventEndFreq,elevel,measDist,
eirp,MeerKATBand,polarisation
{0} "rfi401","recv001",30,40,110,10,50,"None","V"
{1} "rfi401","recv001",80,70,87,10,50,"None","V"
{2} "rfi401","recv001",90,90,80,10,50,"None","V"
{3} "rfi401","recv001",150,200,67,10,50,"HERA","V"
{4} "rfi402","recv002",0,30,60,10,50,"None","V"
{5} "rfi402","recv002",30,70,40,10,50,"None","V"
{6} "rfi403","recv003",30,300,57,50,10,"HERA","V"
{7} "rfi404","recv001",300,400,37,3,50,"None","V"
{8} "rfi404","recv001",756,756,35,3,50,"UHF|L-Band","V"
{9} "rfi404","recv001",1607,1607,35,3,50,"L-Band|S-Band","V"
```

FIGURE 5.4: Sample array database.

### 5.1.3 Implementation of alternative design

The alternative design is created using a single data model, i.e., the relational model. We model key-value data, initially stored in Accumulo, into relational data. Similarly, we transform array data originally stored in SciDB into relational data. As a result, many data objects (Table 5.5) and relationships have been established, as discussed in chapter 4.

TABLE 5.5: Illustrates each data object created using a single data model (relational database)

| Object ID | objectName | fields | database ID |
|---|---|---|---|
| 1 | rfi.attribute | attributeNo, attributeName, dataType, description, objectID | 0 |
| 2 | rfi.engine | engineID, engineName, host, port, connectionProperties | 0 |
| 3 | rfi.database | databaseID, databaseName, engineID | 0 |
| 4 | rfi.object | objectID, objectName, databaseID | 0 |
| 5 | rfi.receiver | receiverID, receiverName, nChannel, bitstream, nAccs, adcType, spectrumBits, rfGain, rxbandwidth, lowFreq, highFreq | 0 |
| 6 | rfi.rxLocation | rxLocationID, receiverID, rxLocationName, rxDirection | 0 |
| 7 | rfi.transmitter | transmitterID, txType, category, txDescription, txNotes | 0 |
| 8 | rfi.intentional | transmitterID, txType, category, txDescription, txNotes, status, startFreq, centFreq, endFreq, txbandwidth, txlatitude, txlongitude, txLocation | 0 |
| 9 | rfi.culprit | transmitterID, txType, category, txDescription, manufacturer, model, serialNumber | 0 |
| 10 | rfi.unknown | transmitterID, txType, category, txDescription, txNotes | 0 |
| 11 | rfi.testmeasurement | testNo, transmitterID, testFacility, testDate, reportDate, reportURL, reportID, reportComp, reportNotes | 0 |
| 12 | rfi.testLocation | testLocationID, testNo, testLocationName, testLongitude, testLatitude | 0 |
| 13 | rfi.permit | permitID, transmitterID rfiNotice, issueDate, expiryDate, usage, permitType, deployDate, permitNotes | 0 |
| 14 | rfi.contactperson | personID permitID, contactName, contactOrganisation, contactEmail | 0 |
| 15 | rfi.restriction | permitID, restrictionNo, restrictionLimited, restrictionUnlimited, rfiZoneMap, useDayOrNight, useBeforeTime, useAfterTime) | 0 |
| 16 | rfi.event | eventID, receiverID, transmitterID, eventStartFreq, eventEndFreq, elevel, measDist, eirp, MeerKATBand | 0 |
| 17 | rfi.occupancy | occupancyID, testNo, month, nights, evenings, strong, moderate, weak | 0 |
| 18 | rfi.schemas | objectID, ObjectName, fields, databaseID | 0 |

Table 5.5 illustrates all RFI data objects under a single data model (relational model), which is implemented in the relational database RFI_catalog_data (databaseID = 0). The implementation of the alternative model differs from that in Table 5.3 in that it consists of more data objects. These additional objects are purposely created to break down the numerous fields into manageable relational tables while still maintaining a consistent relationship. These additional objects include rxLocation, testLocation, contactPerson, and restriction. Additionally, the culprit, intentional, and unknown objects must be created as single entities in the relational database, despite inheriting from the transmitter.

Furthermore, we created each object with its attributes, along with their respective data types. We specify various constraints for each attribute and relationship: PRIMARY KEY – indicates an attribute as a unique identifier in a specific entity. NOT NULL – specifies that the attribute cannot have a null value. ON DELETE CASCADE – used on foreign keys to indicate an action when a record is deleted from a parent's table. For instance, ON DELETE CASCADE implies that all related records in the child's table should be deleted when a corresponding record in a parent's entity is deleted. This is done to maintain data consistency between the two objects.

We provide the relational schema used in the creation of the alternative database

model, which includes all RFI data objects under a single data model implemented in PostgreSQL. Please refer to Appendix E for the full relational schema.

## 5.2 Experimental Design

The experiments ran in a setup typical of the RFI monitoring environment with core applications: monitoring and detection, and data storage. The setup supports the CRUD database operations, enabling both loading new RFI data during monitoring and detection, as well as searches for detailed analyses of RFI. RFI data can be updated or deleted from the database.

The database can be accessed locally or remotely through the RESTful API, using the cURL (client URL) tool [195]. REST is a generic set of constraints (such as having a client/server relationship, and providing a uniform interface) applied to resources in a distributed system [196]. The cURL tool is a command-line tool and library for transferring data using server URLs. cURL supports a wide range of these protocols (e.g., FTP, HTTP, SCP, SSL, SMTP, and TELNET). We use cURL to execute the CRUD operations using the HTTP protocol. Clients access the database through an integrated API (Figure 5.5). cURL tool enables Clients to post data through the URL onto the database server (192.168.0.117) on HTTP port 8080. The database uses the information stored in the islands to direct the query to the respective engines.

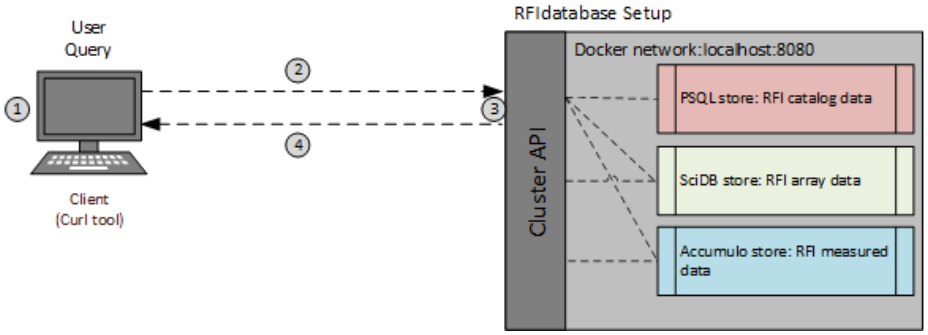



FIGURE 5.5: Experimental setup of the integrated RFI database based on the polystore implementation framework [130]. The parameters measured are: (1) response time, (2) upload speed, (3) latency and (4) download speed.

The database cluster consisted of three clients (PCs) and one host server. The client PCs operated *macOS 10.12*, 16 GB of RAM, 1.8 GHz processing speed, and 500 GB of disk capacity. The host server operated on Ubuntu 20.04.3 LTS, 25 GB RAM, 2.2 GHz processor speed, and 11 TB disk capacity. The cluster provided an average speed of 100 Mbps to and from the server. We tested the performance of both bulk uploads and downloads.

Data is organized in CSV files to enable fast loading of bulk files into the database systems. Recall, that we do not store RFI data in CSV due to its inability to quickly index across multiple files, lack of support for structure-like queries, and poor visualization. The rest parameters are response time, latency, upload speed, and download speed (Figure 5.5 and Table 5.6). These include response time, latency, upload speed, and download speed.

The query response time is the time taken for a query to return the results. The upload speed is the average speed for uploading data, whereas the download speed refers to the average speed of downloading data from the database. We measured

TABLE 5.6: Test parameters for the database implementation.

| Label | Parameter | Description |
|---|---|---|
| 1 | response time | total time for the query to return results |
| 2 | upload speed | average speed for a complete upload |
| 3. | latency | time taken before the actual data transfer begins |
| 4 | download speed | average speed for a complete download |

upload and download speeds after completing upload and download operations. We measured latency from the start of a query operation until the data transfer was about to begin [195, 166]. It includes the connection time and processing time at the database server.

### 5.2.1 Test Parameters

We measured key parameters of database performance: download speed (KB/s), response time (seconds), and latency (seconds). We also measured the speed of loading bulk data files into the RFI database and used cURL to execute these measurements. Download speed refers to the average download speed that cURL measures for the complete download or data transfer [197]. The download speed determines the speed of downloading RFI data from the database server to the clients. This speed is measured by instructing cURL to write out its parameter value by specifying (*'speed_download'*) (see Figure 5.6).

Upload speed refers to the average upload speed that cURL measures for the complete upload [197]. The upload speed determines the speed of uploading RFI data from the clients to the database server and is measured by specifying (*'speed_upload'*) in our cURL evaluation scripts (see Figure 5.6). Our speeds were returned in bytes per second, which we computed as kilobytes per second. Possible errors of these parameters included speeds recorded as either zero or extreme values (skewed).

The query response time is the time taken for the query to return the results (download) from the database. This parameter determines the time a query lasts before it returns the results [197]. It is measured from the start until the client has sent a FINISH (FIN) packet. FIN packets indicate that all the bytes have been downloaded. The response time is measured by specifying (*'time_total'*) in the cURL script (see Figure 5.6). The possible error of this parameter is that time_total still returns a value even though the resource (URL) at the server is not specified.

Database latency is the delay before the data transfer begins (also pretransfer time) [195]. This is measured from the start of the query operation until the data transfer is about to begin. This time includes connection and processing time at the server [166]. We specified (*'time_pretransfer'*) in our cURL statement to return latency in seconds (see Figure 5.6) and measured database latency to determine the ability of our model to handle real-time applications. A possible source of error is when (*'time_pretransfer'*) is recorded as zero even though many bytes were downloaded. This implies zero database latency, i.e., as the query arrives at the database, the download becomes immediately available for transfer. This is unlikely, as the database model will experience either processing time or other delays from the cluster. Besides, there could be other delays that (*'time_pretransfer'*) excludes, especially during the time that transfer is just about to begin. Lastly, we excluded any test with possible errors from our analysis and instead repeated that specific test.

```
curl -X POST -d "bdarray(filter(project(events,
eventID,transmitterID,startFreq,endFreq,elevel,MeerKATBand),
i <= 50 and MeerKATBand <> 'None'));"
http://192.168.0.117:8080/bigdawg/query/ -w
"speed_download": "%speed_download",
"time_pretransfer": "%time_pretransfer",
"time_total": "%time_total"
```

FIGURE 5.6: Sample evaluation query to return the data along with the parameter values of download speed, pretransfer time and response time.

Figure 5.6 is an evaluation query that returns the data and parameter values of response time, download speed, and pretransfer time. Here, the cURL statement projected only eventID, txt, startFreq, endFreq, and MeerKAT band from the events array, of which the first 50 events are detected from a non-MeerKAT band. cURL wrote out data (using *–write-out* or *-w*) in standard output (stdout) after a completed POST transaction to the database server via *http://192.168.0.117:8080/bigdawg/query*. The standard output from cURL is plain text containing the download and parameter values that were sorted, using the grep command [198], and exported to JupyterLab (python notebook) [199] for analysis.

To measure the loading or ingestion speed, we loaded a data file of size 1.2 MB. We measured the duration (loading time) to load the data fully into each data store (PSQL, Accumulo, and SciDB). Also, we computed the loading speed (measured in MB/second) of each store. We loaded, deleted, and recreated the entire data object for every fresh data loading to ensure no reuse of the existing schema. We repeated this sequence for 15 intervals as we collected loading time in seconds. It is important to note that each store has a unique loading technique. Thus it was necessary to test how each store was affected and the database model overall.

### 5.2.2 Query Design

TABLE 5.7: Showing the seven test queries (Q1 to Q7), their categories and explanation.

| Query no. | Category | Query | Explanation |
|---|---|---|---|
| Q1 | Simple | Return all RFI events in a given band or time period | This seeks to understand the existing RFI in a given frequency or at a particular time frame at the site |
| Q2 | Complex | For a particular RFI event detected, show all culprit transmitters and their related permit details | This tracks common culprits with permit details such as permit validity, and permitted zones. This is helpful when investigating whether the permitted transmitter is adhering to the restrictions. |
| Q3 | Complex, join | Return all RFI data objects together with their associated attributes | The query seeks to understand the RFI metadata by listing each object with all associated attributes. This is crucial in keeping the structure consistent while working in multiple data stores. |
| Q4 | Complex, aggregate | Compute the number of events detected in a given MeerKAT band | This determines the most frequently affected MeerKAT band. This can be used to generate frequency plots to guide astronomers during observational measurements. |
| Q5 | Complex | Return all RFI events whose transmission is unknown | The query assists astronomers to build related information about the unknown RFI to be cross-matched with the RFI database |
| Q6 | Cross-island | moves data from one store and display it in another | this query enable astronomers to view all types of data stored across multiple stores |
| Q7 | Cross-island | joins data stored in two distinct stores and display as one view | this query enable astronomers to view RFI data along with metadata in a single view |

We tested seven queries, listed in Table 5.7. We categorize queries as simple, complex, aggregate, and join. A simple query fetches data from a single store, while a complex query fetches data from more than one store. An aggregate query computes and returns a resultant value, while a join query consists of a join transaction that coordinates related data from two or more data objects.

A standard API is set up on the docker containers to be accessed remotely or locally. We tested the API's ability to pick data from a native data model (native-island) and across different models (cross-island). A native island of information relates to a single data model, whereas a cross-island relates to several data models (Section 3.4). Our design has three islands: relational, array, and text-based.

```
Curl -X POST -d "bdtext(
{'op' : 'scan', 'table' : 'transmitter', 'range' :
{'start' : ['rfi001','',''], 'end' : ['rfi004','',''']} });"
http://192.168.0.117:8080/bigdawg/query/
```

(A)

```
Curl -X POST -d "bdrel(
SELECT * FROM bdcast(
bdarray(filter(culprit_measurement,i<=6)), resultant,
'(culpritNo int64, testNo int, startFreq double,
endFreq double, band string)', relational))"
http://localhost:8080/bigdawg/query/
```

(B)

FIGURE 5.7: A) Native island, and B) Cross-island query

Figure 5.7 (A) is a native-island query that returns all transmitters and their related permit details, such as issue and expiry date. Here, the text island is linked directly to the Accumulo (text) store engine that scans the transmitter data object. Related RFI data is arranged chronologically in columns using unique identifiers (rfi001 and rfi004). The query returns all RFI data with keys between rfi001 and rfi004 with their associated values. Also, it computes the total time to complete the entire transaction.

Figure 5.7 (B) is a cross-island query that returns each transmitter along with measurement details if the transmitter type is a culprit. The information is used in assessing the test measurement carried out on each transmitter before it is issued a permit and allowed on the site. The query will first filter measurement array data from the culprit, then cast the results into a relational table showing the culprit number, test number, start frequency, end frequency, and frequency band. Here, the query utilizes two different islands (array and relational).

## 5.3 Test Environments

We test five test environments for our experiments: increasing data volumes, multiple users, varying workloads, bulk uploads, and API environments (native and third-party APIs). The test environments emulate a data-intensive environment at the MeerKAT/SKA radio telescope during RFI monitoring. We examined the impact of each environment on the RFI database model. We measure query response time (seconds), upload and download speeds (KB/seconds), and latency (seconds) across the three data stores: PostgreSQL (PSQL), SciDB, and Accumulo. Each query was performed 15 times, and the median value recorded excluded outliers.

### 5.3.1 Increasing Data Records/Volumes

Data volumes are likely to double rather than increase linearly at the SKA/MeerKAT radio telescope [4]. This test examines how an exponential increase in data records or volume affects the performance of the RFI database model. The number of records is doubled per experiment, starting from 200 records to 25600 records in each of the three stores (PSQL, SciDB, and Accumulo). For each system, we initially loaded 200 records, followed by the individual queries retrieving all the 200 records in each store, as we measured response time latency and speed. Subsequently, we deleted the 200 records before doubling the loading to 400 and repeated this process

incrementally up to 25,600 records. We achieve the doubling using test data from a data tool [200].

Our upload queries differed in sizes per store: SciDB (33 bytes), PSQL (65 bytes), and Accumulo (118 bytes). Similarly, the data records stored in each data store differ in size: 25600 records in SciDB amounted to 1.7 MB (average 66 bytes per record); in PSQL they amounted to 1.5 MB (58 bytes per record), and in Accumulo to 1 MB (39 bytes per record). The data records in Accumulo were small due to the key-value structure (i.e., a unique key and a value that consisted of a few bytes), whereas SciDB and PSQL have a tabular structure. Data records greater than 25600 in a single query affect the system, particularly the PQSL and Accumulo data stores, thus returning no results. Therefore, we considered a maximum of 25600 records in a single query transaction to avoid several timeouts with no results.

### 5.3.2 Multi-user Environment

The multi-user tests emulate several concurrent users accessing the RFI database simultaneously. The initial test served as a baseline with a single user operating on one of the three client PCs, ensuring that no other active users accessed the database concurrently. In the second test, we considered a total of three concurrent users, with one user utilizing each of the three PCs. The third test involved six concurrent users, with two users actively accessing the database on each client PC. The fourth test involved twelve concurrent users, four on each client PC. In the fifth test, we tested a total of eighteen, with six at each client PC. The sixth test had twenty-four concurrent users, eight users to each client PC. The seventh test involved thirty concurrent users, ten on each client PC. Lastly, in the eighth test, we considered a total of thirty-six concurrent users, with twelve on each client PC.

We implemented an increase in the user load to simulate the typical growth rate observed in most organizations [4]. We adjusted the number of concurrent user requests in our executing script (see Appendix I) that we ran on each client PC. Recall, that our setup consists of three client PCs and one database server (Section 5.2).

Each user requested 1 MB in data queries. Our work focused on bulk queries, typical of our monitoring environment use case. At maximum, there were 36 users (36 MB of data) in a single session. This number is sufficient for testing the database model as it would support 36 concurrent monitoring devices at a specific time. We examine the impact per user within the database cluster to provide performance estimates of response time, latency, download, and upload speed for more than 36 users.

### 5.3.3 Varying workloads

Varying workload is typical of the SKA environment with different user needs. For instance, the catalog database that stores the RFI metadata requires frequent Reads but fewer Updates. Each user must read the metadata to enable easy use of the other data stores in the cluster. We define database workloads that consist of different proportions of Inserts, Reads, and Updates. We investigate how the database responds to a combination of query operations. This test is motivated by the YCSB benchmarking standard recommended for evaluating large and scalable systems [201, 157]. The standard considers a mix of two database operations categorized as varying workloads (see Table 5.8).

TABLE 5.8: YCSB workloads under consideration

| | | |
|---|---|---|
| Workload A | 50% Reads | 50% Updates |
| Workload B | 95% Reads | 5% Updates |
| Workload C | 100% Reads | 0% Updates |
| Workload D | 95% Reads | 5% Inserts |

We defined four workloads: Workload A - 50% Reads and 50% Updates; Workload B - 95% Reads and 5% Updates; Workload C - 100% Reads Only; and Workload D - 95% Reads and 5% Inserts. In Workload A, we measure the impact of reading 50% and updating 50% of the data records in each data store. Workload D examines the impact of 95% Reads and 5% insertions of new data records into the database. Workload C was the reference Workload used to evaluate Workloads A, B, and D [162, 160].

Workload A Query targeting PSQL will retrieve or read 50% of the total amount of metadata stored in PSQL, while the other 50% is updated. Workload B Query targeting PSQL will retrieve/read 95% of the metadata, while the other 5% is updated. Workload C Query targeting PSQL will retrieve/read 100% of the metadata with no other database operation running concurrently. Workload D Query targeting PSQL will retrieve/read 95% of the metadata with 5% metadata being inserted. The same workload specification is applied for Accumulo and SciDB, targeting key-value data and array data, respectively, as we measure response time, latency, upload, and download speed. In varying workloads, download speed, for example, for Workload A Queries, refers to the speed it takes to retrieve 50% and update 50% of the data in either PSQL, Accumulo, or SciDB. Meanwhile, upload speed is the speed at which Workload A requests are sent to the server.

This work does not focus on Workload E (Short ranges) and F (Read-Modify) on the YSCB benchmark. Workload E does not represent bulk queries on which this work focuses. Workload F is not supported across the three data stores of the RFI database model. These workloads under study represent the frequency of user needs at the MeerKAT/SKA radio telescope. We focused on several Reads (>95%) and Updates (> 50%) as they are recommended for evaluating scalable databases [104].

### 5.3.4 Bulk uploads

This environment measures bulk ingestion of the database – loading bulk RFI data files into appropriate data stores. Bulk loading is a common way to enter large data files into a database. For a data file of size 1.2 MB for each data store (PSQL, SciDB, and Accumulo), we measured the loading average speed and loading time (time taken to complete the loading of a specific data file). We format data in CSV format and write scripts to load the data files into respective data stores as we measure speed and time. This measurement is essential for the RFI monitoring environment, which requires the fast loading of bulk datasets into the database. Moreover, it is one of the key requirements for new and scalable databases [104].

### 5.3.5 API Environment

External users, such as collaborators from regions other than SKA/MeerKAT, will communicate with the database via web applications (third-party) rather than the native API. APIs are essential, specifically in distributed environments (such as the

SKA international project), to map and access quickly data from appropriate data stores [152].

The API environment tests the native API against a third-party API, *Insomnia* [202]. The native API is inherent to the framework, and the third-party API is an application that runs on top of the RFI database. Insomnia is a REST (Representational State Transfer) API that uses HTTP methods to access web resources using URL-encoded parameters. Insomnia is open-source, with low response times. Also, it organizes better workflows than the more popular APIs, such as Postman [203]. Moreover, Insomnia supports a wide range of API protocols, the cURL (client URL) tool inclusive, which is integral to the polystore design and plugs in well with the framework to minimize overheads.

In this test, we investigated how this database API impacts the performance of the RFI database. We ran a similar query in a native API and a third-party API (Insomnia) at specific times as we measured response time. Each API fetched the same data from the three data stores using the four classes of queries: simple, complex, aggregate, and join.

### 5.3.6 Single Database Environment (Relational)

In this test environment, we model all RFI data to fit within a single data model. The purpose is to evaluate the impact of storing RFI data in different formats within a unified data model, specifically the relational model. We opt for testing the traditional relational model, which has a proven track record spanning over five decades and has demonstrated superior performance compared to newer models, such as key-value and array data models, in certain environments [102, 104]. Consequently, we structure key values and multidimensional arrays as relational tables implemented in a relational database system (PSQL). We ensure that the database environment accommodates all diverse RFI data while increasing the number of concurrent users from 1 to 36. Each user requests bulk queries to retrieve approximately 1 MB of data from the database as we measure response time and latency. The objective is to examine the impact of RFI key-value data and array data on a relational database to determine whether it is suitable to retain RFI data in its native data structures that dictate the storage.

### 5.3.7 Cross-island Environment

This test environment assesses the impact of transferring 1 MB of data between various storage systems within the cluster. The primary objective is to evaluate how these cross-storage queries affect the RFI database model. Four types of data transfers are considered: PSQLToAccumulo, AccumuloToPSQL, PSQLToSciDB, and SciDBToPSQL.

The PSQLToAccumulo query involves retrieving 1 MB of RFI data from PSQL and transferring it to Accumulo. Conversely, the AccumuloToPSQL query transfers the same amount of data from Accumulo to PSQL. Similarly, the PSQLToSciDB query is configured to move 1 MB of RFI data from PSQL to SciDB, while the SciDBToPSQL query transfers 1 MB of data from SciDB to PSQL. These queries empower scientists with the capability to seamlessly transfer data across different storage systems within the database cluster. This, not only accelerates scientific analysis but also facilitates a comprehensive understanding of the complex phenomena under investigation. Therefore, it is essential to evaluate the impact of moving bulk data across different storage settings on the overall database model.

In addition, we model a cross-island join query to retrieve data from two distinct stores. We write a procedural program whose output includes results from both PSQL and SciDB queries, based on a common attribute (see Appendix C.7). We consider 'SciDB_with_metadata' to indicate a query that returns 1 MB of array data in SciDB and metadata in PSQL, while 'SciDB_without_metadata' indicates a query that returns the same array data while excluding the metadata.

The program's model involves the 'SciDB_with_metadata' query issuing two commands in a procedural approach. The first task entails executing an array query, followed by checking if the returned attributes exist in the metadata stored in the PSQL store. If a match is found, we output the metadata associated with each attribute included in the array data, along with the array data (see Appendix C.7 for sample output), as we measure execution time in seconds. The existence of attributes in both stores is a key factor in linking the data from the two stores. Therefore, we evaluate the performance of the 'SciDB_with_metadata' query against 'SciDB_without_metadata' in a multi-user environment. This test is crucial for scientists who often work with complex phenomena that require the rapid definition/description of several attributes and parameters involved in their complex datasets. Similarly, this approach can serve as the initial stage in developing more efficient query processing schemes for join queries in polystore environments, which are still a challenge [131].

# Chapter 6

# Results and Discussion

In this chapter, we measure the performance of the RFI database model in a range of test environments, including increasing data volumes, multiple users accessing the data concurrently, varying workloads, bulk loading, and API environments. The RFI database model consists of three data stores: PSQL, SciDB, and Accumulo. Each stores distinct RFI data. Response time is a key performance indicator, as are the speed and latency of individual data stores within the cluster. Recall that the acceptable maximum latency without interruption is one second, while there are as yet no standard performance estimates for upload speed, download speed, and response time for new and scalable databases.

## 6.1 Impact of Data Volumes

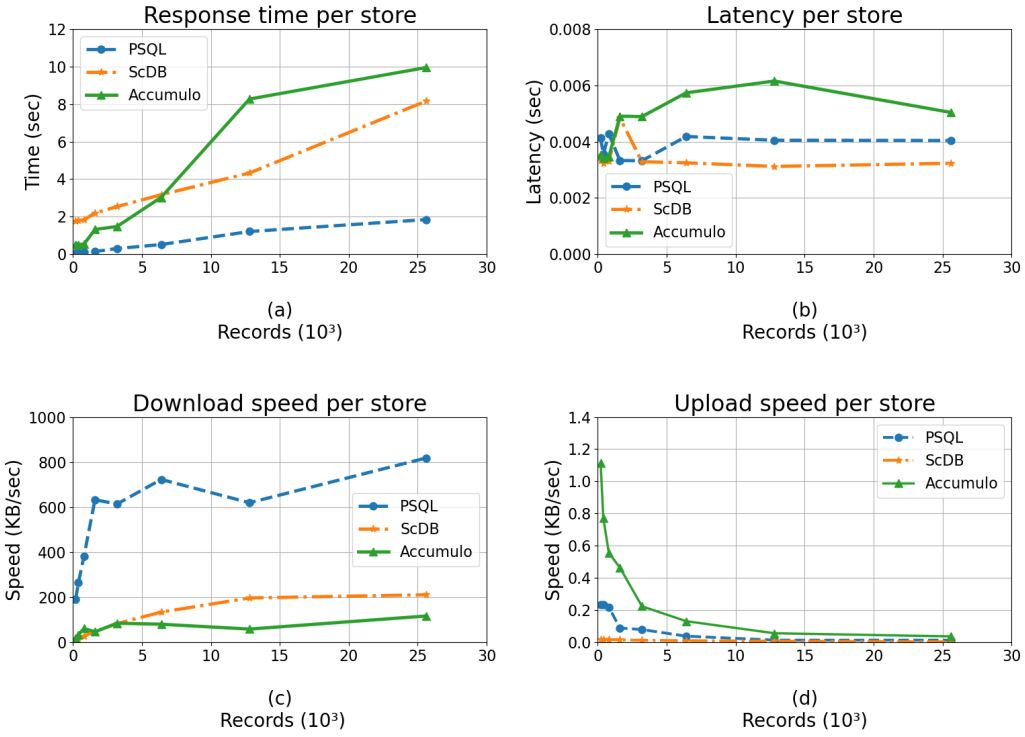



FIGURE 6.1: The impact of an increase in the number of records on the database model: (a) response time, (b) latency, (c) download speed, and (d) upload speed are shown for each data store. Data for each graph is in Appendix D.1.

Figure 6.1(a) plots the median response time with increasing records (doubling from 200 to 25600). For all three stores, this shows an increase in response time as the number of records doubles, as expected. However, the increase in response time for PSQL is much lower than for Accumulo and SciDB: an increase in records has a lower impact on PSQL's response time as compared to Accumulo and SciDB. PSQL queries are faster to connect to a database, thus a shorter response time, while Accumulo and SciDB queries depend on several applications within the cluster. This is in line with previous studies that indicated that PSQL databases have better response times [103, 104, 102].

For up to 6400 records, SciDB has a longer response time, followed by Accumulo and PSQL. Above 6400 records, Accumulo response times increase to about 10 seconds compared to SciDB (8 seconds) and PSQL (under 2 seconds). The sudden rise in response times in Accumulo between 5000 and 15000 records possibly is due to background activities that run in the Accumulo cluster, which may impact the performance of Accumulo [121]. Accumulo and SciDB aim to make data available all the time (high availability) [22, 106]. This is achieved by storing data in multiple stores.

For quick data retrievals from distributed locations, indexes are created in each location, but the use of several indexes affects the response time [204]. For large numbers of records (>12800), Accumulo's response time is greater than SciDB's. Accumulo takes a longer time to scan through each key and its associated data compared to SciDB, which maps a single key to a block of data [117]. While Accumulo's and SciDB's response times are impacted by increasing data volumes, these stores are optimized for data availability, even during a node failure or interruption in the query transaction. The largest portion (about 80%) of RFI data fits into Accumulo and SciDB Stores, whereas 20% of the RFI data (mostly metadata) is in PSQL.

Our measured response times are in line with previous studies that suggested response times of 4 to 20 seconds for returning 10000 to 100000 records [163, 164]. For large numbers of records, Accumulo and SciDB queries would affect the overall response time of the database. Therefore, as the number of records increases, we recommend extending the database setup with additional hardware.

Figure 6.1(b) plots latency with increasing data records. Accumulo has the highest latency, while PSQL has the lowest. For low numbers of records, there are fluctuations in latency for all stores, while for large numbers of records, latency remains approximately constant, as expected [205, 157]. Latency is less than 0.007 seconds, well below the acceptable maximum of one second [168, 157, 185]. Fluctuations in latency across all stores are small, and this should not impact the overall database performance.

Figure 6.1(c) shows download speed with increasing data records. Note that data records differ in size in each store (see Section 5.3.1). PSQL's download speeds are higher than SciDB's and Accumulo's: the minimum download speed in PSQL is roughly the maximum in Accumulo (approx. 200 KB/sec). PSQL's download speed increases from approximately 200 to 600 KB/sec for fewer than 1600 records and to between 600 and 800 KB/sec for > 1600 records, as the number of records increases. Accumulo and SciDB's download speeds increase more gradually and remain reasonably stable for large numbers of records (> 6400). Fluctuations in download speeds are due to congestion caused by high numbers of records transmitted in a short time. This is why at 600 KB/sec PSQL's download speeds begin to fluctuate more than SciDB's and Accumulo's at 200 KB/sec. There is a possibility that with large numbers of records (> 25600), PSQL fluctuations in

download speeds may increase more than SciDB and Accumulo. However, PSQL has the highest download speeds. The higher the download speed the better the store performs. PSQL download speeds, therefore, are better, specifically in small numbers of records (< 1600) than Accumulo's and SciDB's [103, 104]. Previous work has shown SciDB and Accumulo to provide a steady performance with large numbers of records rather than fast downloads [103, 21]. The fast download speeds in PSQL for a small number of data records will benefit SKA scientists during frequent requests for the RFI metadata. Meanwhile, SciDB's and Accumulo's ability to store large data records will be of advantage when handling large astronomical surveys.

Figure 6.1(d) shows query upload speed with increasing records, as expected. The SciDB upload query is 33 bytes compared to PSQL's 65 bytes, and Accumulo's 118 bytes (Section 5.3.1). This is a decreasing trend in the upload speed for all stores as the number of records increases.

The percentage drop in upload speed over the number of records tested for Accumulo is 84%, followed by 15% for PSQL and then 1% for SciDB. Accumulo's and PSQL's upload speeds are more affected by increasing records despite having higher upload speeds than SciDB whose query upload speed stays relatively low and consistent. SciDB's queries are slow to upload, but not affected by an increase in data records. For large numbers of records (> 20000), the upload speed for all stores remains low at 0.05 KB/sec. Low upload speeds across the stores may lead to 'no download' at all – zero bytes downloaded and no data being returned. This is possibly due to slow query connections to the database server resulting in long delays caused by large requests [132]. This, consequently exacerbates the overall response time of the database.

Previous studies have indicated that small numbers of requests are faster to upload than bulk requests [206]. We expect the upload speed of the queries to drop as the number of records increases (see Appendix D.1 (d)). Our model, therefore, is affected by bulk requests but shows a great performance for small requests. More powerful servers are recommended to improve the response speed of bulk queries. Meanwhile, fast query uploads would benefit the MeerKAT/SKA during RFI monitoring, research, and planning future discoveries.

## 6.2 Impact of Multiple Users

We investigated how an increase in users affects the performance of the database model. We considered a maximum of 36 concurrent users, each downloading about 1 MB of data (Figure 6.2).

Figure 6.2(a) graphs the median response time with users from 1 to 36. We observe a linear increase in response time for SciDB and Accumulo, but PSQL shows a response time that is constant at approximately 3 seconds from 12 users onwards. SciDB has the highest response time (4 to 12 seconds), followed by Accumulo ( 1 to 8 seconds), and 1 to 3 seconds for PSQL. The lower the response time as the number of users increases, the better the store performance. PSQL's response times perform well, particularly for more than 12 users, irrespective of the number of users being added to the system. SciDB's and Accumulo's performance are impacted as their response time continues to increase with increasing numbers of users. SciDB and Accumulo lack proper indexing, moreover, their data is stored in different locations within their cluster, thus taking a long time for queries from several users to resolve the location of data. Therefore, as the number of users increases, the

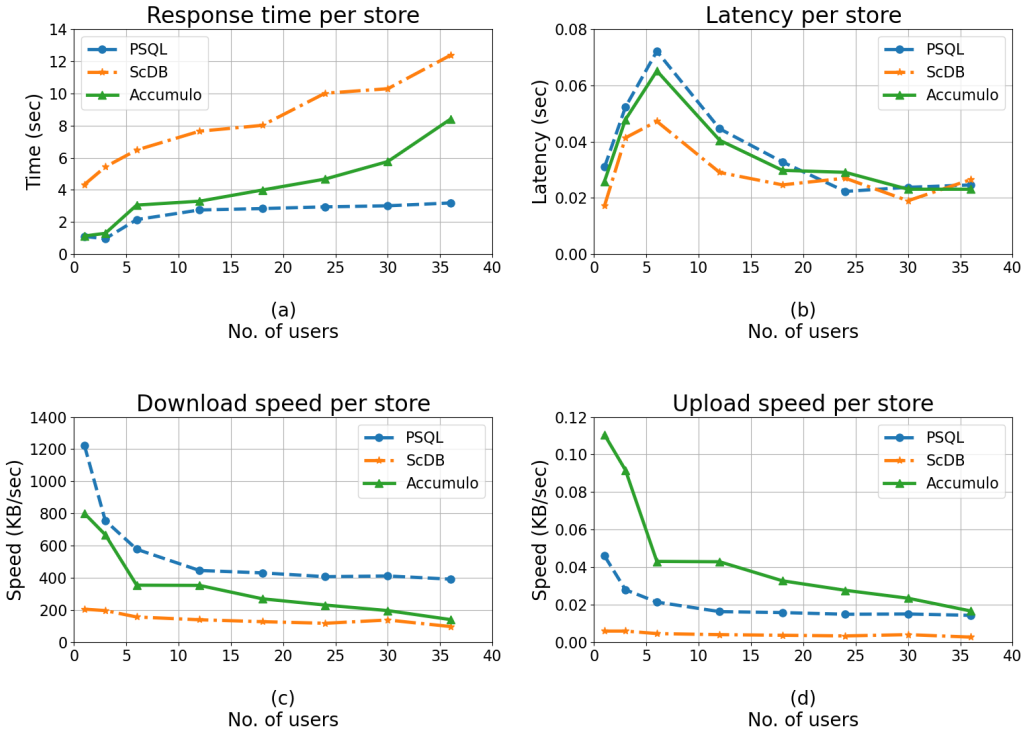



FIGURE 6.2: Impact of multiple users on the RFI database model: (a) response time, (b) latency, (c) download speed, and (d) upload speed are shown for each data store. Data for each graph is in Appendix D.2).

SciDB and Accumulo's response times are expected to increase the overall response time. Additional servers to the cluster would handle the increasing number of users without exacerbating the overall performance.

Figure 6.2(b) shows latency (time) with increasing numbers of users. For PSQL and Accumulo stores, latency increases from 0.03 to 0.07 seconds for 1 to 6 users, and from 0.02 to 0.05 seconds for SciDB. The spike in latency at <= 6 users is due to additional users per store (PSQL, SciDB, and Accumulo), accessing the database concurrently. An extra user introduces an additional processing time on the server. However, beyond 6 users, each database model reuses frequently requested data or existing patterns stored in a buffer of each data store. The patterns reduce the overall latency because the database does not require to access the disk each time the database request is initiated. This is why latency for all stores drops and afterward stays at around 0.02 seconds. Previous studies by Jing et al., [207] have indicated that scalable databases store existing data patterns in memory to reduce the latency and thus improve overall database response time. Despite the increase in latency due to multiple users, our figures are still low and lie well within the generally acceptable latency limits (< 1 second) [157, 205]. Above all, SciDB has the lowest latency among the three stores.

Figure 6.2 (c) shows download speed with increasing numbers of users. The larger the decline in download speed, the greater the impact the number of users has on the database. A steady download speed indicates a less significant impact on the database performance. SciDB has the lowest overall download speed, from 200 KB/sec dropping to 150 KB/sec. An important observation is that each store has

its highest download speed when accessed by a single user.

PSQL flattens off at around 400 KB/sec, the highest across all numbers of users tested. Interestingly, Accumulo's download speeds continue to decline with increasing numbers of users. PSQL provides better download performance for large numbers of users. Accumulo's poor download performance could be due to the several applications that run the Accumulo cluster in that an additional user to the cluster would slow down the download speed, as Accumulo needs to coordinate various background activities for each user [121]. Therefore, downloads consisting of SciDB and Accumulo data impact on overall download performance of the database. However, this can be improved upon by deploying the database model on a high-speed network.

Figure 6.2(d) shows query upload speeds with increasing numbers of users. Upload speed decreases with increasing numbers of users for Accumulo and PSQL. PSQL's upload speed levels to 0.015 KB/seconds, and Accumulo's continues to decrease with an increase in the number of users. SciDB's upload speeds remain relatively flat and, the lowest (< 0.01 KB/seconds) across all numbers of users tested.

The percentage drop in query upload speed as the number of users increases is highest for Accumulo (70%), followed by PSQL (25%) and then SciDB (5%). Accumulo and PSQL queries are more impacted by high numbers of users than SciDB, despite having higher speeds. Unfortunately, based on this graph, for large numbers of users, the upload speed of the model is likely to drop drastically to 'no upload' or zero uploads. This is expected for all data stores as multiple connections create delays in the database. However, very slow uploads affect latency, which consequently exacerbates the overall response time of the database model.

## 6.3 Impact of Varying Workloads

Figure 6.3(a) shows the median response time with varying workloads (A, B, C, and D) for each datastore. Note that workload C (100% Only Reads) represents our reference Workload; Workload A runs 50% Reads and 50% Updates; Workload B runs 95% Reads and 5% Updates; Workload D runs 95% Reads and 5% Inserts (see Section 5.3.3).

Across all the stores, Workload D has the fastest response time, followed closely by Workload B. This implies that Insert operations take longer than Update operations for all stores. Note that the Update operations in Accumulo are Insert operations that overwrite existing data [117, 205], which is why, for Accumulo, these two operations show equivalent response times. Further, the response time for Workload C is longer than that for Workload A for all stores, except for Accumulo, whose response time for Workload A and C is equivalent. It takes less response time to read and update half the workload in A than reading the entire Workload C. Our model's response time is more affected by Workload D, followed by Workload B and Workload A. The model performs better with frequent Reads and Updates than with Insert operations. Frequent Reads and Updates in scalable databases create patterns of existing data stored in memory to speed up access rather than to access data stored on the disk. Database Inserts, on the other hand, consist of new entries yet to be stored in the database, thus difficult to create data patterns [162, 161]. At the SKA telescope, Most RFI applications, such as RFI cross-matching, monitoring, and generating statistics, require frequent Reads with a few Updates and Inserts.

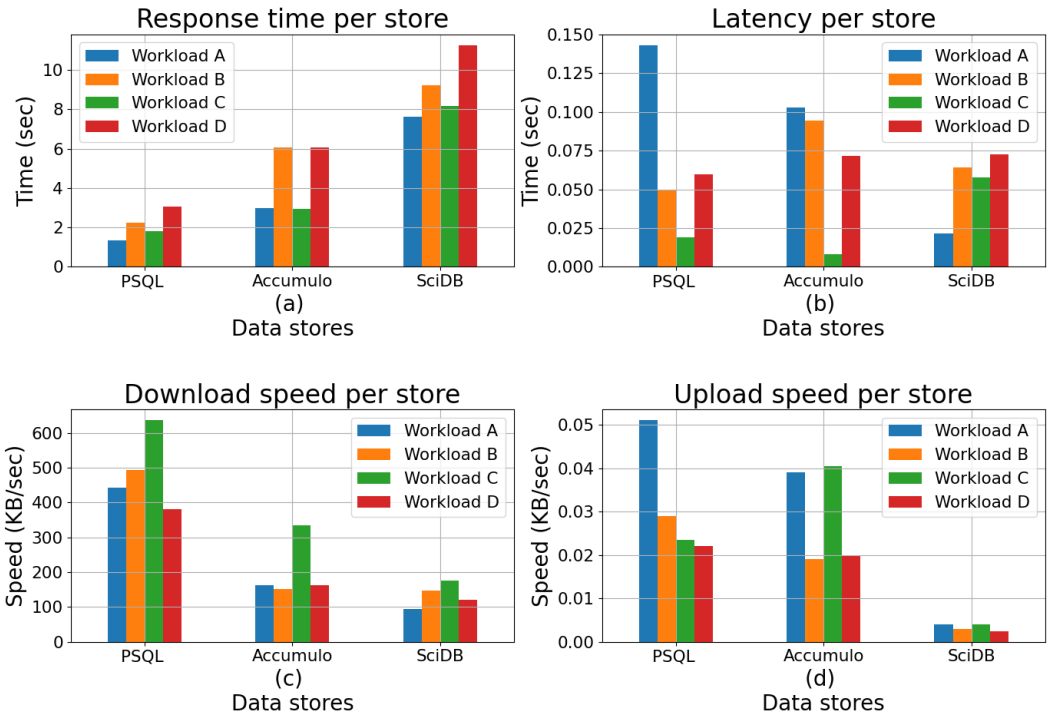



FIGURE 6.3: Impact of varying workloads on the database model. Workload A (Reads: 50%, Updates: 50%); Workload B: (Reads: 95%, Updates: 5%); Workload C: (Reads: 100%); and Workload D (Reads: 95%, Inserts: 5%). Data for each graph is in Appendix D.3.

Figure 6.3(b) shows the latency of workloads A, B, C, and D in each datastore. For PSQL and Accumulo, Workloads A, B, and D have a higher latency than Workload C, while for SciDB, the latency of Workload C is higher than that of Workload A. The latency across the three stores varies from 0.01 to 0.14 seconds across the workloads. Workload A has the highest latency in PSQL and Accumulo but the lowest in SciDB where the highest latency is for Workload D. Workload C has the lowest latency in PSQL and Accumulo. We expect the latency per workload to vary from one store to another. However, it should not exceed one second (i.e., the acceptable maximum latency) [168, 157]. Our model shows that the maximum latency recorded from all the experiments is 0.14 seconds, which meets the acceptable latency standards of the scalable databases.

Figure 6.3(c) shows download speeds for each workload in each store. Workload C has the fastest download speed compared to Workloads A, B, and D. Workloads with Read operations only are faster to download than Read workloads in combination with Inserts or Updates. This is in line with previous studies [161, 162] that have shown better performance for Workload C as they run without interruption from competing Update or Insert operations found in the other workloads. Further, Workload B's download speed is higher than Workloads A and D for PSQL and SciDB but is quite similar to A and D for Accumulo. Recall here that Accumulo Inserts and Updates are very similar operations and so will have very similar speeds as shown in Workload A and D. PSQL has the fastest download speeds for all the Workloads across the board because of the store ability to index data stored in one location quickly.

Overall, our model's download speeds are best for Workloads B and C. In other

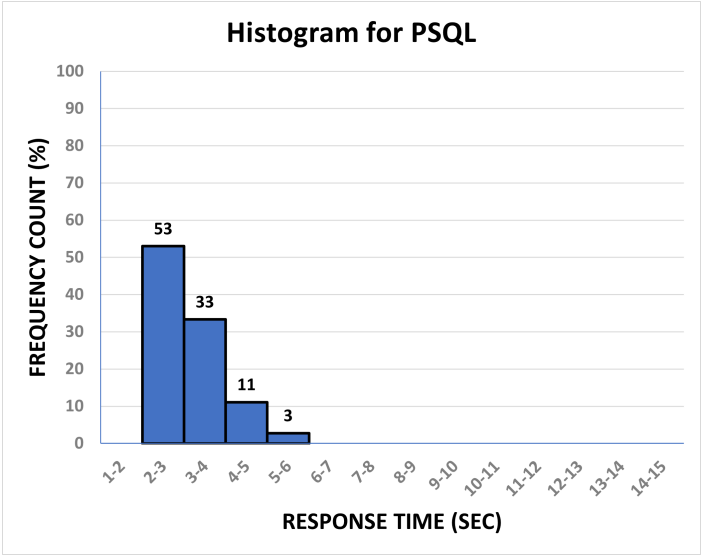
words, more effective on frequent Reads and Updates than Insert operations. Inserts impact the model more than Updates due to the lack of existing data or query patterns from Insert operations. Reads and Updates data patterns are stored in memory for faster retrievals to avoid access to the disk most of the time [162, 161]. In addition, Inserts in scalable databases use distributed partitions/blocks that require more time to replicate data on other blocks. In contrast, Reads and Updates use index/keys to locate the block of data to read or update quickly [113]. Several Inserts are likely to affect the download speed of the database. However, these could be handled better by loading CSV, JSON, or HDF5 data files into the database.

Figure 6.3(d) shows the query upload speed of the various workloads for each data store. For PSQL and SciDB, Workload D has the lowest upload speeds compared to Workloads A, B, and C. For Accumulo, the upload speed of Workload D is slightly higher than Workload B. Recall Updates and Inserts are equivalent, thus the small difference in Workload D and B. The upload speed of Workload C is highest for Accumulo and SciDB, followed by Workload A, and B. PSQL's upload speeds in Workloads A and B are higher than in Workload C. Recall that Workload C consists of Read operations only. PSQL is not impacted by competing Updates but affected by Inserts. On the other hand, Updates and Inserts impact the upload speeds in SciDB and Accumulo. Overall, the model's query upload speeds are best for queries with Reads operations only but are impacted by competing Update and Insert operations.

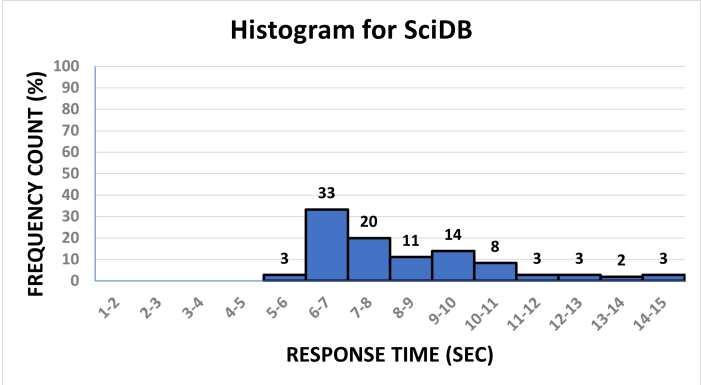
## 6.4 Response Time and Latency Distribution

The distribution of response time and latency enables performance estimates for our model. Figure 6.4 consists of histograms illustrating the distribution of response time in each store across a total of 36 concurrent users. Each histogram presents the frequency count as a percentage against the response time recorded by the users.

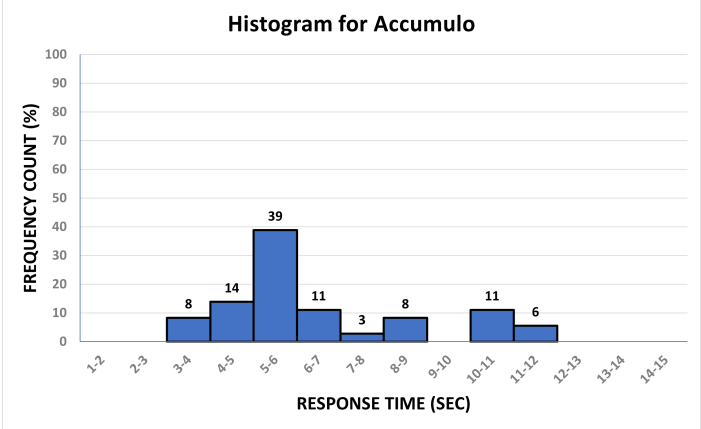
Figure 6.4(A) shows the distribution of response times for PSQL. The response time ranges from 2 to 6 seconds, SciDB 5 to 15 seconds (Figure 6.4)(B), and Accumulo 3 to 12 seconds (Figure 6.4)(C). PSQL has a narrow distribution in response times, while SciDB and Accumulo show wide distributions. The distribution in PSQL response times appears to be positively skewed, as the frequencies decline away from the peak. This implies that smaller values are more common than larger values in our dataset. On the other hand, the distribution in SciDB and Accumulo's response times appears to be right-skewed, indicating that the majority of values are found on the left-hand side of the peak.




(A)




(B)




(C)

FIGURE 6.4: The distribution of response time for multiple users across three stores: A) PSQL, B) SciDB, and C) Accumulo.

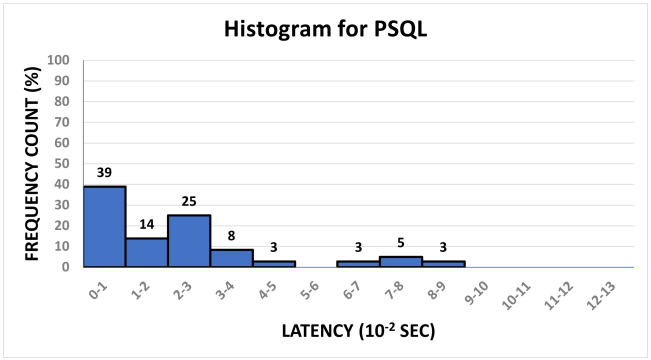
For PSQL, the average response time is 3.06, and the median is 2.955 seconds. The most frequently occurring response times for users in PSQL lie within the 2 to 3 seconds range (modal class), representing 53% of the users. SciDB's average is 8.41 seconds, and the median is 7.89 seconds, with a modal class of 6 to 7 seconds representing 33% of the users. Meanwhile, Accumulo's average is 6.53 seconds, and

the median is 5.76 seconds, with a modal class of 5 to 6 seconds representing 39% of the users.

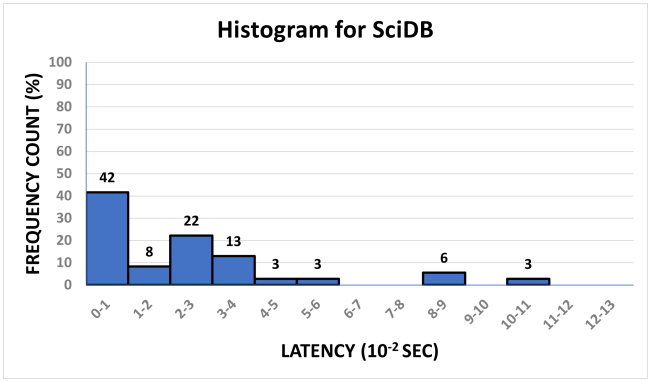
In all stores, the average and median are observed to fall either within or near the modal class, suggesting the majority of the values are near or within these modal classes. For instance, 86% of users in PSQL experience response times within 2 and 4 seconds, and in SciDB, 53% of users encounter times between 6 and 8 seconds, while in Accumulo, a similar 53% experience times within 4 and 6 seconds. These percentages enable us to model response time estimates for users in each store.

Therefore, our model predicts that the highest number of users in PSQL will experience response times between 2 and 4 seconds, in SciDB between 6 and 8 seconds, and in Accumulo between 4 and 6 seconds. These estimates serve as performance benchmarks for our model and for evaluating the efficiency of new and scalable databases.

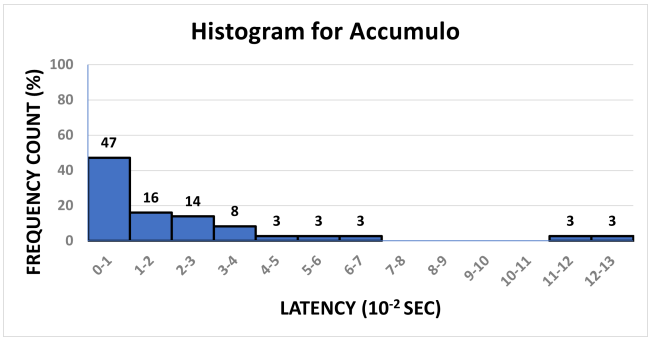
Figure 6.5(A) shows the PSQL's distribution of latency. The distribution range in PSQL is 0.09 seconds, 0.11 seconds for SciDB (6.5(B)), and 0.13 seconds for Accumulo (6.5(C)). The distribution of latency in all stores seems to be right-skewed, as the frequencies decrease away from the peak. It is evident in all figures that there are few occurrences of larger response times (> 0.04 seconds), while the majority of the values appear under 0.04 seconds.




(A)




(B)



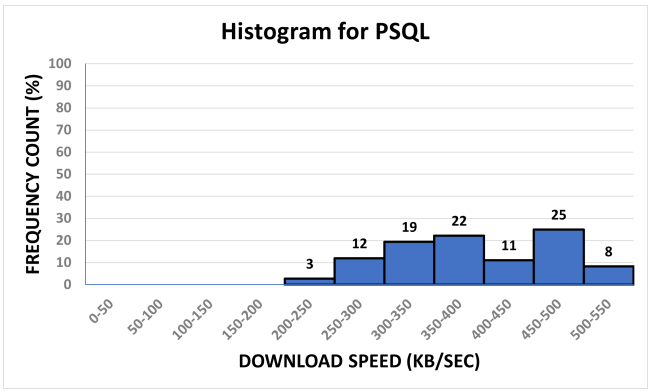

(C)

FIGURE 6.5: The distribution of latency for multiple users across three stores: A) PSQL, B) SciDB, and C) Accumulo.

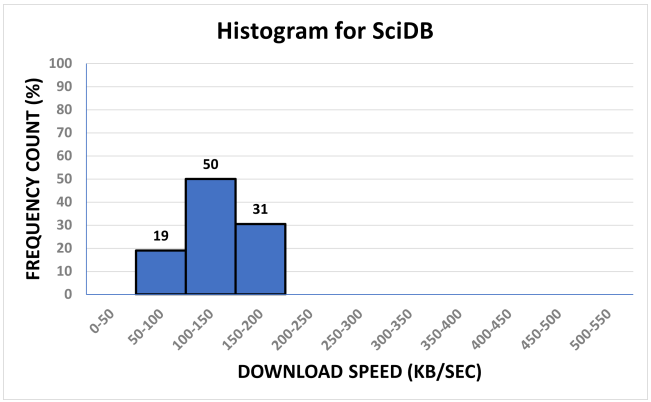
All the stores have the same modal class (between 0 and 0.01 seconds). This means about 40% of users in each store experience latency under 0.01 seconds. Meanwhile, about 85% of users have a latency of fewer than 0.04 seconds, and about 15% of users experience latency greater than 0.04 seconds. High latency can be a result of poor database connections and server interruptions from concurrent users. This is expected in a multi-user environment (such as the MeerKAT/SKA radio telescope), where many users with varying data needs connect to the database concurrently. Fortunately, our latency values across all experiments lie well within the acceptable maximum limit (less than 1 second). Our model, therefore, ensures a user of low latency, irrespective of their data needs.

## 6.5 Download and Upload Speed Distribution

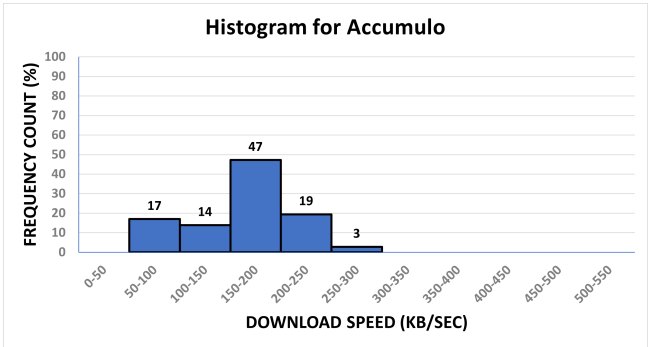
Figure 6.6a(A) shows the PSQL's distribution of download speeds. The download speed distribution in PSQL ranges from 200 to 550 (range = 350) KB/sec, SciDB's range 150 KB/sec (Figure 6.6b)(B), and 250 KB/sec for Accumulo (Figure 6.6c)(C). PSQL and Accumulo have a wider distribution compared to SciDB.




(A)




(B)




(C)

FIGURE 6.6: The distribution of download speed for multiple users across three stores: A) PSQL, B) SciDB, and C) Accumulo.

For PSQL, the average download speed is 397 KB/sec, and the median is 393 KB/sec. Accumulo's average is 164 KB/seconds, and the median is 168 KB/seconds, while SciDB's average is 126 KB/seconds, and the median is 127 KB/seconds. PSQL's modal class is (450 - 500 KB/sec) accounting for 25% of the users, SciDB's is (100 - 150 KB/sec) with 50%, and Accumulo's is (150 - 200 KB/sec) for 47% of users. This suggests that SciDB has the highest number of users with similar download

speeds, followed by Accumulo, and then PSQL, despite PSQL having the fastest speeds.

The majority of the data in SciDB appears in a narrow range of 100-200 KB/sec, representing 81% of the users, compared to PSQL and Accumulo. Specifically, PSQL's range is 250-500 KB/sec, representing 89% of users, and SciDB's range is 50-250 KB/sec, representing 97% of users. These estimates provide performance benchmarks for estimating query download speed for bulk users requesting about 1 MB from the database. Additionally, previous studies do not show performance standards for download speeds in new and scalable databases [102, 160, 162].

Figure 6.7a(A) shows the distribution of query upload speeds for PSQL. The query upload speeds in PSQL range from 0.0075 to 0.020 KB/sec, in SciDB from 0.0025 to 0.010 KB/sec (Figure 6.7b)(B), and in Accumulo from 0.010 to 0.0325 KB/sec (Figure 6.7c)(C). This implies that SciDB has a narrow distribution in upload speeds, while SciDB and Accumulo show wide distributions.

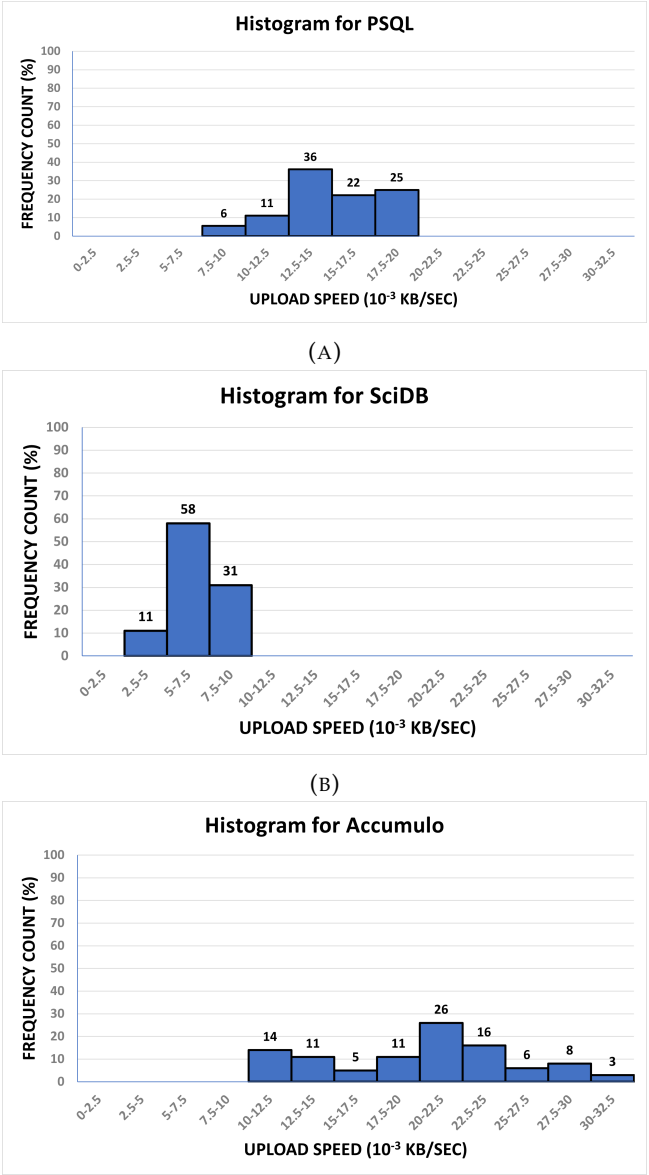



FIGURE 6.7: The distribution of upload speed for multiple users across three stores: A) PSQL, B) SciDB, and C) Accumulo.

For PSQL, the average upload speed is 0.0145 KB/sec, and the median is 0.014 KB/sec. SciDB's average is 0.00375 KB/second, and the median is 0.004 KB/second, while Accumulo's average is 0.0195 KB/second, and the median is 0.02 KB/second. PSQL's modal class is (0.0125 - 0.015 KB/sec), accounting for 36% of the users; SciDB's is (0.005 - 0.0075 KB/sec) with 58%; and Accumulo's is (0.020 - 0.0225 KB/sec) for 26% of users. This suggests that SciDB has the highest number of users with similar download speeds, followed by PSQL, and then Accumulo. Note that PSQL and Accumulo have a higher percentage of users with speeds greater than 0.010 KB/sec than SciDB, which has all users concentrated below 0.010 KB/sec.

As observed in the histograms, Accumulo has overall faster upload speeds with a wider range than PSQL and SciDB. Accumulo transmits more bytes per second, followed by PSQL and SciDB. Note the upload query sizes in Accumulo (118 bytes), PSQL (65 bytes), and SciDB (33 bytes). Approximately 94% of users in PSQL have similar upload speeds between 0.010 and 0.020 KB/sec. In contrast, 89% of SciDB's

users experience speeds between 0.005 and 0.010 KB/sec, whereas 83% of users in Accumulo encounter speeds between 0.010 and 0.025 KB/sec. These results show that the highest number of users have upload speeds between 0.005 and 0.025 KB/sec, which is adequate considering the small-sized (< 118 bytes) upload query requests retrieving about 1 MB of data. In reality, upload queries from a good database are expected to consist of as few chunks of data as possible to enable faster execution of the upload queries/requests [103, 140]

## 6.6 Bulk Uploads

Loading bulk data files into the database had a median loading time of 324.30 seconds for 1.2 MB of RFI data into Accumulo over a set of 15 uploads. The data file loaded into PSQL and SciDB has much shorter loading times: PSQL 0.16 seconds and SciDB 0.12 seconds. SciDB had the fastest loading speed of about 10 MB/sec, followed by PSQL with 7.5 MB/sec, and Accumulo with the slowest at 0.004 MB/sec. Key-value data (in Accumulo) takes more time to be loaded into the database than relational (PSQL) and array data (SciDB). PSQL and SciDB use CSV files to support bulk loading into respective data structures. In contrast, Accumulo is less optimized for bulk loading, as inserting a record requires rewriting all the column/table files, thus, Writes are not effective in Accumulo [113, 117].

Moreover, Accumulo uses the conventional ETL (extract, transform, and load) techniques, which involve extracting raw data, formatting in sorted key-value pairs, and loading into a target destination. This technique is less suited for loading bulk data. Studies at MIT [121] have used SuperCloud computing facilities to support the bulk loading in Accumulo. Our model takes much longer to load key-value data into the Accumulo than relational and array data stored in PSQL and SciDB stores, respectively. Overall, it is fast to load bulk RFI data, particularly relational and array data using loading techniques in CSV, JSON, and HDF5 files rather than several Insert operations.

## 6.7 Database APIs

Figure 6.8 shows the response time of the native API against a third-party API (Insomnia RESTful) with low response times (between 0.1 and 1 second) [202].

The response times are very similar in both APIs for different queries. However, the third-party API has slightly higher average response times than the native database API. There is an insignificant increase of 5% in response time for the third-party API for all queries, except the aggregate query where the native and the third-party APIs have the same times. For a simple query, the native API takes 0.57 seconds, while the third-party API takes 0.61 seconds to retrieve the same data from the database. A native API is expected to perform slightly better as it resides in the database cluster, unlike a third-party API.

The response times for Join queries are much higher than simple, aggregate, and complex queries. Joins incur a longer response time to execute, for our model that stores data in multiple nodes/locations [194, 117]. However, there is a small difference (1.34 to 1.41 seconds) between the native and third-party API response times under Join queries. The aggregate queries have the same response times of 0.59 seconds for both native and third-party APIs. An equivalent performance in the APIs implies that there is no significant impact associated with an aggregate query in both APIs. Aggregate queries involve computing a single resultant value from

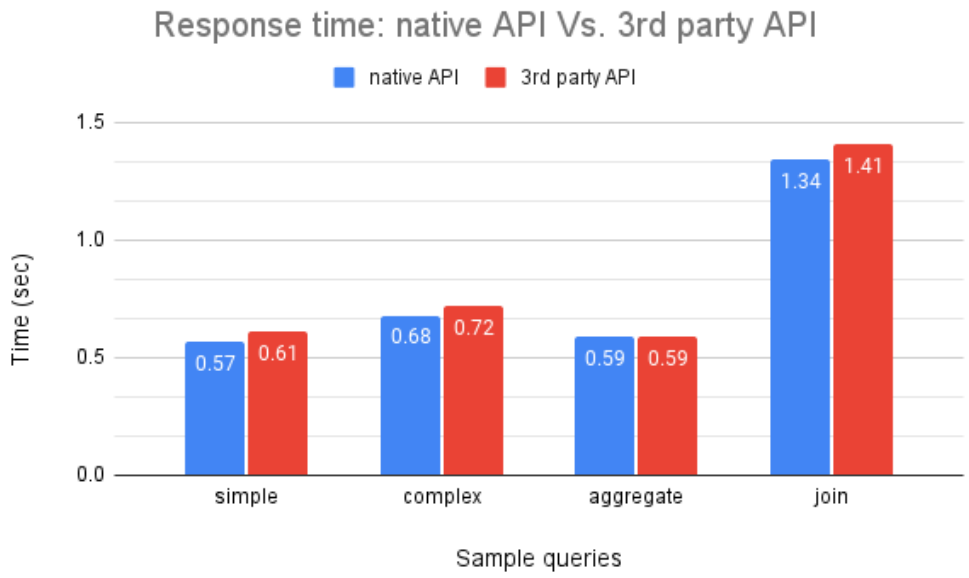



FIGURE 6.8: Showing response time of native vs. third-party API (Insomnia) connected to the RFI database.

aggregate functions such as *MAX()*, and *AVG()* [204, 194, 184]. Therefore, returning such a single value from two APIs would not impact the response time. This is why our results indicate similar performance for both APIs.

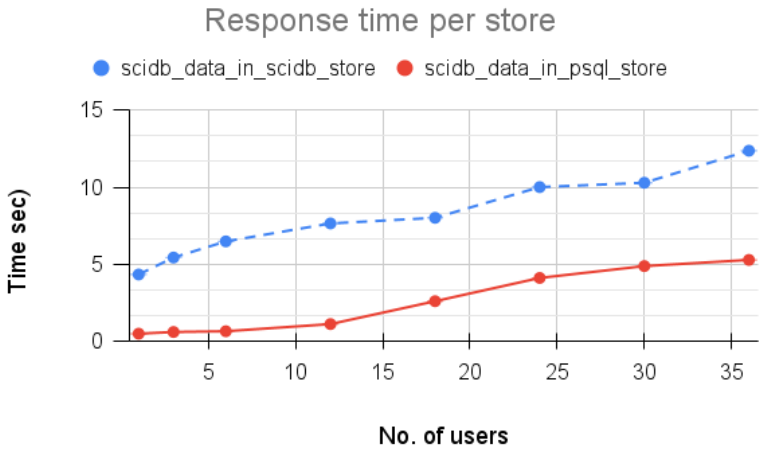
Overall, our results suggest that Join queries are costly to run, whereas simple, complex, and aggregate queries are easy and faster as they require fewer database executions.

## 6.8 Impact of SciDB data on PSQL store

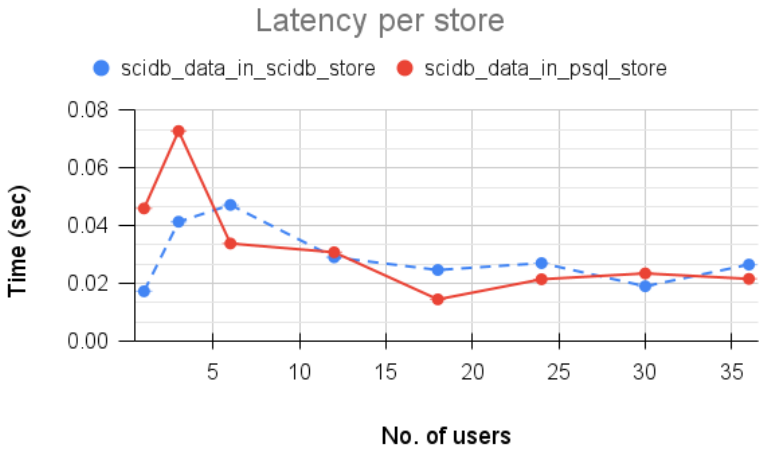
Figure 6.9 shows the performance impact of SciDB data in two different store environments: 1) SciDB Store indicated 'scidb_data_in_scidb_store' and 2) PostgreSQL (PSQL) indicated by 'scidb_data_in_psql_store' as the number of users increases.

Figure 6.9a shows the response time increases with an increase in user load. As the number of users increases, the response times for both store settings also increase. This is expected in real-world systems, such as database applications [164], where an increase in the number of users leads to more database requests, causing contention at the server, which subsequently affects the response time.

One key observation is that 'scidb_data_in_scidb_store' generally exhibits longer response times than 'scidb_data_in_psql_store'. This indicates that PSQL outperforms the SciDB store when SciDB data is stored in PSQL compared to when SciDB data is stored in SciDB. This implies that RFI array data in SciDB if moved to the PSQL store, would result in significantly improved response times compared to when it is stored in SciDB. This is attributable to the structure of array data in SciDB that is easily transformable into relational tables. In addition, SciDB is built based on traditional relational technology, which means that array data would thrive in relational tables (PSQL store) [22, 147]. However, previous studies [103, 104, 102] have shown PSQL to deteriorate with large data sets, such as a huge volume of sequential array data with multiple dimensions.




(A)




(B)

FIGURE 6.9: Impact of SciDB data on PSQL store: (A) response speed, and (B) latency across two different environments, i.e., SciDB ('scidb_data_in_scidb_store') and PSQL ('scidb_data_in_psql_store').

The increase in response times for up to 24 users is significantly lower in 'scidb_data_in_psql_store' compared to 'scidb_data_in_scidb_store' as the number of users increases. Beyond 24 users, both store settings continue to experience growth, with SciDB showing a more rapid increase in response times compared to PSQL. This suggests that 'scidb_data_in_psql_store' may be a more scalable choice, particularly for a high number of users (e.g., 30 and 36 users), as it consistently maintains lower response times. This difference can be attributed to PSQL's ability to perform well under lighter user loads compared to SciDB. In SciDB, multiple applications are required to run an operation regardless of the load size [21, 147]. Consequently, this reliance on other applications affects SciDB's response time.

Furthermore, the gap in response times between the two store settings significantly increases with an increasing number of users. This may be a result of both stores exhibiting potential scalability challenges. Therefore, our results are crucial when considering the scalability requirements of your application.

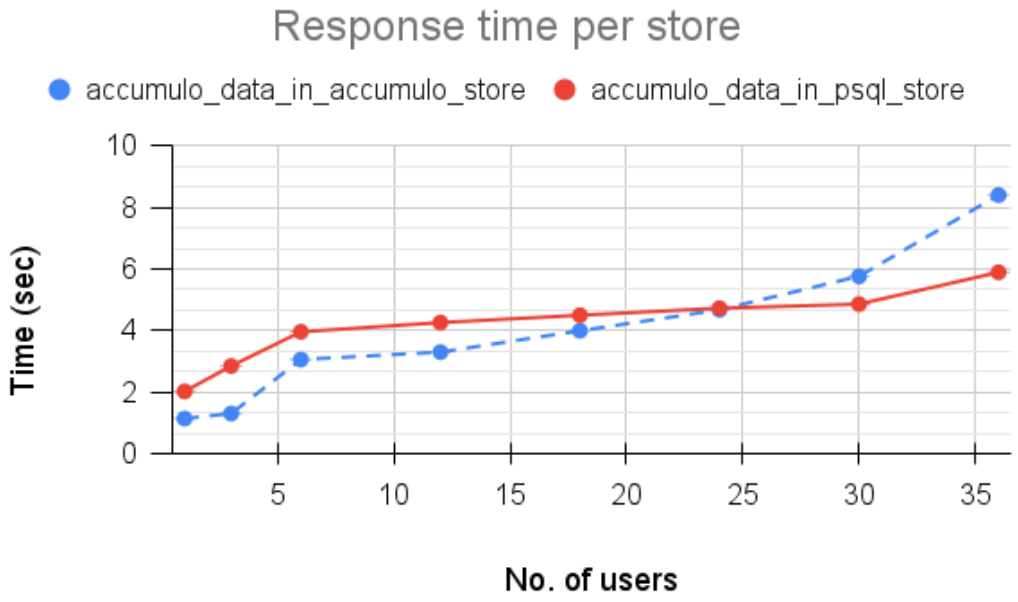
Figure 6.9b illustrates the latency values for 'scidb_data_in_scidb_store' and 'scidb_data_in_psql_store' storage settings. The latency values for both settings

are consistently low, ranging from approximately 0.014 seconds to 0.073 seconds. These values indicate low for our model which in turn results in quick quick data retrieval and responses across increasing user load. This latency pattern aligns with real-world scenarios, which often involve consistent delays of less than 1 second [168, 157].

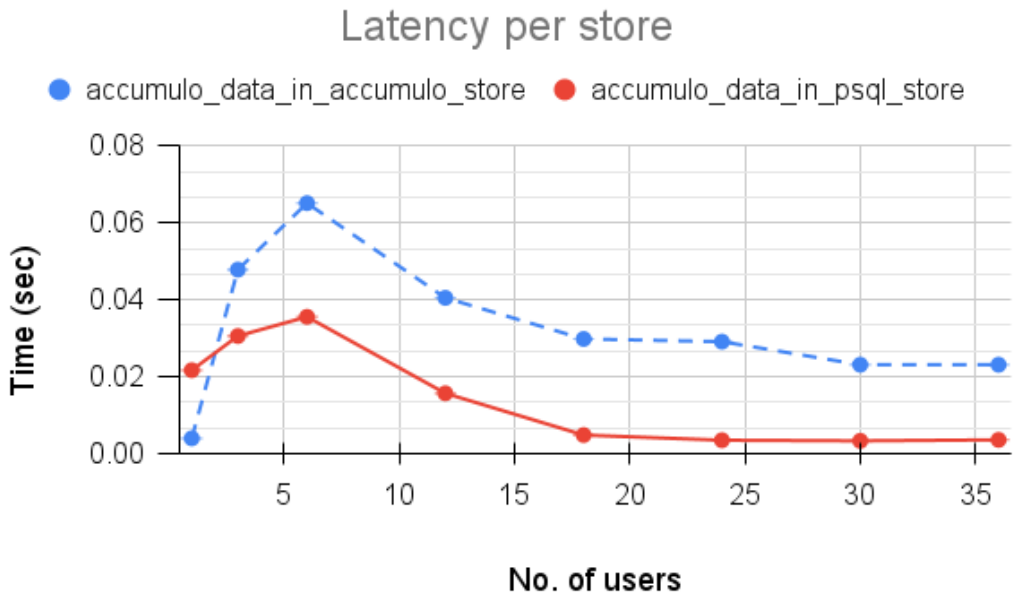
We observe higher latency for scidb_data_in_psql_store than scidb_data_in_scidb_store as the number of users grows from 1 to 6. Beyond 6 users, scidb_data_in_psql_store's latency drops significantly to a minimum of 0.14 seconds, while scidb_data_in_scidb_store's latency remains higher than scidb_data_in_psql_store's at higher user loads, except at 30 users. In general, despite the variation in latency exhibited in both store settings with an increasing number of users, their latencies consistently stay below 0.075 seconds. At a higher number of users, they stabilize at around 0.02 seconds. This implies that both store settings experience relatively low and stable latencies. The stability in latency issues at higher user loads may be attributed to buffers that have a standardized data limit for sending data to the database server. Typically, at higher loads, the buffers fill up to the maximum and work with a specific transfer rate to and from the server [208].

## 6.9 Impact of Accumulo data on PSQL store

Figure 6.10 illustrates the performance impact of Accumulo data in two different store environments: 1) the Accumulo store, indicated by 'accumulo_data_in_accumulo_store', and 2) PostgreSQL (PSQL), indicated by 'accumulo_data_in_psql_store', across various user loads.




(A)




(B)

FIGURE 6.10: Impact of Accumulo data on PSQL store: (A) response speed, and (B) latency across two different environments, i.e., Accumulo ("accumulo_data_in_accumulo_store") and PSQL ("accumulo_data_in_psql_store").

Figure 6.10a depicts response times as the number of users increases in two storage settings: "accumulo_data_in_accumulo_store" and "accumulo_data_in_psql_store". With an increasing number of users, both storage configurations exhibit a rise in response times. The notable increase in response times as the number of users grows serves as a clear indication of the impact of user load on response times.

For user loads up to 24 users, the response times in 'accumulo_data_in_psql_store' are consistently higher, ranging from 2.02 seconds for 1 user to 4.72 seconds for 24 users, compared to the response times in 'accumulo_data_in_accumulo_store' which range from 1.14 seconds for 1 user to 4.67 seconds for 24 users. However, beyond 24 users, the response times in 'accumulo_data_in_accumulo_store' experience a rapid increase, reaching 8.40 seconds at 36 users, while the response times in 'accumulo_data_in_psql_store' remain relatively low, at around 5.88 seconds for 36 users.

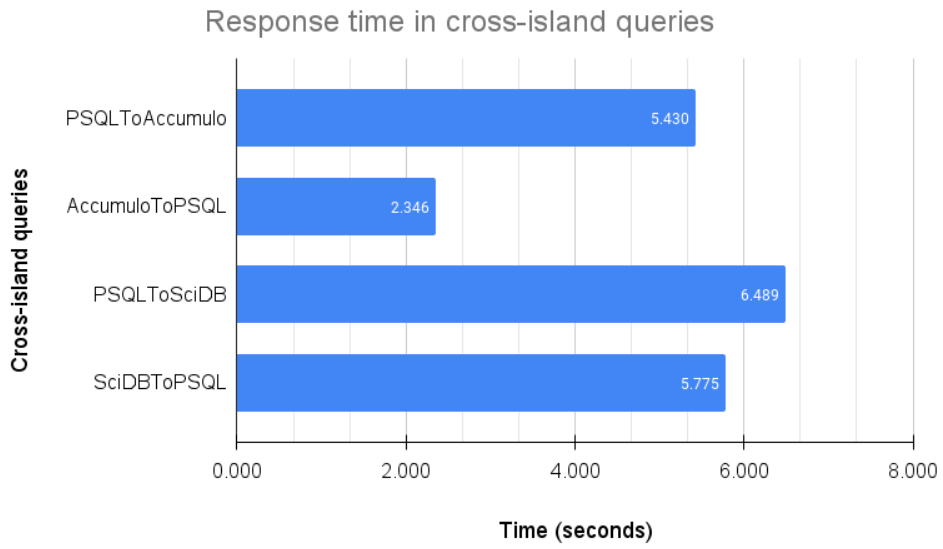
Overall, the data indicates that an increase in user load affects response times. This effect is more pronounced for 'accumulo_data_in_psql_store' at lower user loads but less significant for higher numbers of users (> 24 users), where the response times remain relatively low compared to the more rapid increases in response time for 'accumulo_data_in_psql_store' The sharp increase in response time after 24 users may be a result of potential scalability issues with the model. Therefore, these thresholds are fundamental when implementing scalability in different store settings. In this case, Accumulo scales better when storing Accumulo (key-value) data with fewer users (less than 24 users), while PSQL would scale much better when storing Accumulo RFI data with a higher number of concurrent users (above 24 users) anticipated.

Figure 6.10b shows the latency values for 'accumulo_data_in_accumulo_store' and 'accumulo_data_in_psql_store' storage environments. As the number of users increases from 1 to 36, we observe varying latency values for both storage settings. We notice a gradual increase in latency for 'accumulo_data_in_accumulo_store', starting as low as 0.004 seconds and reaching a maximum of 0.65 seconds for 1 and 6 users, respectively. However, after 6 users, the latency in this setting significantly drops and stabilizes at around 0.003 seconds at higher user loads.

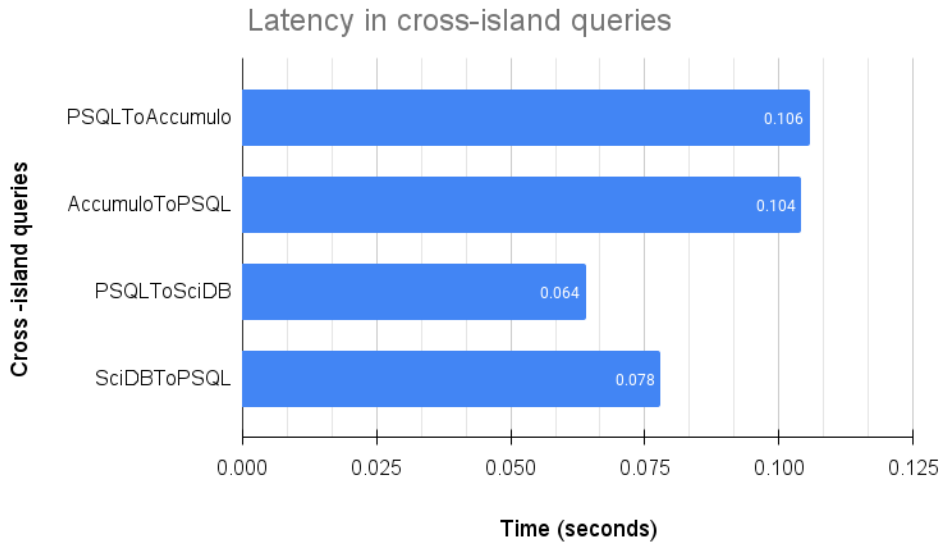
In contrast, the latency values for 'accumulo_data_in_psql_store' start very low at 0.022 seconds for 1 user and gradually increase as the number of users grows, reaching a maximum of 0.035 seconds after 6 users. However, after 6 users, the latency values for this setting drop to as low as about 0.003 seconds, even with 36 users. Generally, the latency values in 'accumulo_data_in_accumulo_store' remain slightly higher than in 'accumulo_data_in_psql_store'. Similarly, the latency values for both settings decline to lower and consistently stable values. For example, beyond 16 users, 'accumulo_data_in_accumulo_store' values stabilize at around 0.025 seconds, while 'accumulo_data_in_psql_store' stabilizes at about 0.003 seconds. The stability in latency values, particularly within the maximum acceptable values in both storage settings at higher user loads, suggests efficient and stable performance, making both settings ideal for quick data retrieval.

## 6.10 Impact of cross-island queries

Figure 6.11a shows response time values for different types of queries, each involving 1 MB of data transfer between various database models, as regarded as cross-islands queries. These response times are measured in seconds, and they represent the time it takes to complete these queries.




(A)




(B)

FIGURE 6.11: The analysis focuses on two aspects: (A) response time and (B) latency of cross-island queries. These queries, namely PSQLToAccumulo, AccumuloToPSQL, PSQLToSciDB, and SciDBToPSQL, involve fetching 1 MB of RFI data from one storage to another.

We observe that AccumuloToSciDB queries have the shortest transfer time, approximately 2.346 seconds, for querying 1 MB of data from Accumulo to SciDB. This is followed by PSQLToAccumulo, which takes 5.430 seconds, then SciDBToPSQL with 5.775 seconds, and finally, PSQLToSciDB with the highest time, around 6.489 seconds. These observations highlight the relative efficiency of querying data from Accumulo to PSQL, resulting in the quickest response time.

In comparison, AccumuloToSciDB queries are faster than PSQLToAccumulo queries. This suggests that querying 1 MB of Accumulo data to a PSQL database is quicker than moving 1 MB of PSQL data to an Accumulo database. The delay is

incurred during the translation semantics involved when casting the intermediate query results from one data model to another. Similarly, it is faster to query 1 MB of SciDB data in PSQL than to query the same amount of data from PSQL to SciDB. It's worth noting that casting data from the SciDB or Accumulo models to the PSQL data model is faster, while it is more time-consuming for the PSQL relational model to cast data to the other data models. The intermediate query result to be cast is organized in a row fashion, which leverages PSQL queries [105, 102]. On the other hand, it would take much longer to translate the row-oriented intermediate result into arrays and key-value pairs found in SciDB and Accumulo, respectively.

Figure 6.11b displays latency values for various cross-island queries, each involving the retrieval of 1 MB of data from one data store to another. The PSQLToSciDB query has the shortest latency of about 0.064 seconds, followed by SciDBToPSQL with 0.078 seconds, then AccumuloToPSQL with 0.104 seconds, and lastly, PSQLToAccumulo with 0.106 seconds. In comparison, PSQLToSciDB exhibits the lowest latencies compared to SciDBToPSQL. This indicates that PSQLToSciDB queries are processed more quickly at the database server compared to SciDBToPSQL queries. On the other hand, PSQLToAccumulo queries have nearly identical latencies, taking approximately the same amount of time at the server before the actual data is transferred to the client.

Overall, these latency values for the cross-island queries closely resemble each other, especially when comparing them to their reverse queries. Therefore, the difference between these queries is relatively low and consistent. This suggests that querying data across different data models does not have a significant impact on the overall performance of the database, making it suitable for various data storage environments.

## 6.11 Cross-island Join query performance

Figure 6.12 illustrates the performance impact of a join query retrieving data from PSQL (stores metadata) and SciDB (stores array data). We measure the response time of a join query, indicated by 'SciDB_with_metadata', and without a join, indicated by 'SciDB_without_metadata', in a multi-user environment. These two test scenarios are described in Chapter 5.

In both scenarios, the response time increases with an increase in the number of users. However, the increase in 'SciDB_with_metadata' is generally higher than when metadata is not included (SciDB_without_metadata). This is expected, as the query output in 'SciDB_with_metadata' is greater than in 'SciDB_without_metadata'. In other words, a SciDB query with metadata returns a larger result compared to a query when metadata is excluded. Consequently, large query sizes affect the overall response time of the database model.

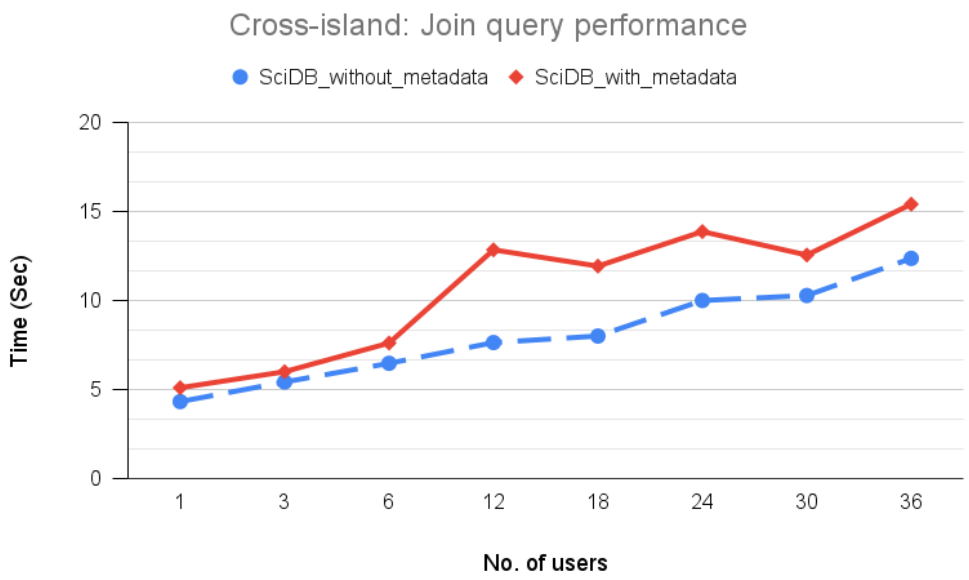



FIGURE 6.12: Impact of a cross-island join query on response time in a polystore environment: With join (SciDB_without_metadata in Blue) and without join (SciDB_with_metadata in Red).

Despite a linear increase, we notice that for a smaller number of users (< 6 users), the performance difference is minimal compared to when the number of users is beyond 6 users. This implies that the increase in response time in 'SciDB_without_metadata' is relatively stable compared to the increase in 'SciDB_with_metadata'. 'SciDB_with_metadata's response times are unstable in higher numbers of users (beyond 6 users), which results in, at times, very high response times compared to cases without the metadata. This is a potential sign that 'SciDB_with_metadata' experiences scalability issues as the number of users increases. Therefore, our join query is less effective with a growing number of users.

Our cross-island join query (SciDB_with_metadata) consists of a sequence of statements or procedures (see Appendix C.7), instructing the compiler on 'what to do' and 'how to do it,' essentially procedural, compared to a non-procedural query (SciDB_without_metadata) that only instructs 'what to do' [183]. The join algorithm in the procedural case is implemented in Java, creating a data abstraction since it is external to the DBMS. In contrast, the non-procedural case uses an inherently sophisticated join algorithm to speed up data retrieval. This is why the execution of procedural queries takes longer than declarative or non-procedural queries, making them less effective in the polystore environment. Our results align with previous studies by Gadepally et al [131], suggesting that more work needs to be done in better processing and optimization of join queries in a polystore system to scale more efficiently.

## 6.12 Discussion

Response times are in line with previous studies [163, 164] that have indicated similar figures for bulk requests, thus indicating acceptable response times. We found a low response time: the model can run bulk downloads of 1 MB of data from several users in under 12 seconds. This will benefit scientists at the MeerKAT/SKA

during RFI monitoring and detection of new RFI. These require quick searches and, therefore, call for a low response time of the RFI database. This achievement would not only benefit scientists at MeerKAT/SKA during RFI monitoring and detection but also other applications, such as healthcare, which deal with large volumes of data and demand quick responses. Storing and quickly retrieving healthcare data, including thousands of patients' records, medical images, reports, and genomic data, is a significant challenge, especially for timely drug development and medical research [209, 210]. If our model is applied in healthcare, it would provide efficient storage for various medical datasets. For instance, patient historical data could be stored as structured data in a relational database, and medical images could be managed using a key-value structure, with the keys stored in the database and the actual images stored externally. Storing numerous images within a database can adversely affect its response time [183]. Other applications, such as genomic data, which typically collects data in a sequential structure, are best suited for array-based data stores. Therefore, our model would ensure timely decisions when requesting diverse medical data from different storage sources.

We found reasonably consistent latency across the three data stores (PSQL, SciDB, and Accumulo). The latency of our model is low (less than 0.14 seconds) and well below the maximum limit (one second) of a scalable database [168]. Also, our model suggests that 85% of users under a multi-user environment experience latency below 0.04 seconds. This is in line with previous studies that have indicated similar latency figures [168, 185]. Our model, therefore, ensures low latency as users request different RFI data in tables, arrays, and documents. These low latency values are important for minimizing the impact on the overall response time of the database, given that they are a tiny fraction of the response time. These low-latency values are crucial for delay-sensitive applications like telecommunications, medical, military and defense systems [209, 182, 211]. These systems require minimum delay as they connect and communicate with other subsystems. Therefore, If applied in telecommunications, our model would facilitate quick connections with other communication devices, thereby enhancing the overall quality of service (QoS). QoS is a mechanism that guarantees delay-sensitive applications a high extent of performance [212]. QoS is a significant concern, particularly as the number of users increases on the database cluster or network with limited bandwidth, leading to congestion that can degrade service quality [211]. Similarly, when applied in military and defense systems, our model will ensure fast communication as they query a variety of data, including profiles, maps, documents, audio, text, reports, and videos, during various operations. Therefore, maintaining low latency is a core requirement in these applications, especially when dealing with numerous subsystems, such as polystore systems.

Our results have indicated that the database model performs well for heavy Reads, followed by Updates and Inserts. In contrast, heavy Insert operations affect the database. Therefore, our model is effective on Reads and Updates, whereas several Insert operations can be avoided by suggesting the use of loading techniques (see Section 5.3.4). Experimental studies, such as [162, 161], have indicated a similar performance pattern for scalable databases. At the MeerKAT/SKA telescope, heavy Updates or Inserts are less frequent compared to Reads. Most RFI applications, such as cross-matching RFI, flagging, monitoring and detection, as well as reporting, require mostly reading from the database, followed by Updates and Inserts. SKA/MeerKAT applications, such as on-site monitoring and the detection of RFI signals, require multiple reads from the database within a short time frame. Similarly, the local spectrum management body (ICASA in South Africa) requires

fast database reads to support continuous monitoring of the limited RF spectrum, ensuring its optimal utilization [213]. Currently, there is a risk of encroachment into bands reserved for radio astronomy, space research and earth exploration, military and defense [213, 214], resulting in interference. Therefore, regulatory bodies can efficiently integrate the RFI database model into their database to continuously monitor spectrum frequency utilization, easily locating all legally and illegally transmitting devices.

We found a decline in download speed with increasing numbers of users. This is expected as more users on the database cluster result in a decrease in transmission speed. Similarly, upload speed decreases with increasing numbers of records or users. Model queries are of small sizes (<118 bytes) that should not impact the database. The small-size queries execute faster than bulk requests, as found in [206]. Besides, the maximum speed of the database cluster (100 KB/sec) can handle several uploads/requests consisting of small data chunks. Moreover, the SKA transmission speeds are expected to improve the performance of the database model over high-speed connections of about 4 TB/sec [100]. High speeds are a key requirement to the SKA international telescope project as it becomes fully operational [86, 84].

We found that it is fast to load bulk data into the database, particularly relational data (in PSQL) and array data (in SciDB), except for key-value data (in Accumulo). The relational and array data have a tabular structure supported by several file applications such as CSV, JSON, and HDF5 files [17, 21]. Key-value structures handle loading/Inserts line by line compared to tables and arrays that load several Inserts directly in their respective data objects [113]. The loading of key-value data objects impacts the speed of bulk uploads of the database model. Therefore, loading unstructured RFI data, record by record would impact the performance of the database model. Our model would effectively handle bulk loading using CSV, JSON, and HDF5 files. To improve the loading speed of bulk data of the model, we suggest using high computing facilities to speed up several Inserts [121]. This is beneficial to the SKA monitoring environment, which requires fast loading of bulk data into the database – also expected as one of the purposes for new and scalable databases [104]. Therefore, If the model is applied to data-intensive projects like the LSST telescope, it would greatly benefit. The LSST telescope generates approximately 15 TB of data daily and requires a database that can support high data input in the shortest period possible [26]. Bulk loading has been a challenge in traditional database settings; therefore, integrating our model would facilitate the speedy loading of data into the database. Moreover, telescope data is naturally collected in CSV, HDF5, or JSON files, which are compatible with the loading techniques used in the model. It's worth noting that the provision of high-performance computing facilities in the future, would significantly improve the speed of bulk uploads in several data-intensive applications.

Our results have shown similar performance by the native API and the third-party API across various queries such as simple, complex, aggregate, and Join queries. While a third-party API is external to the database and adds a new level of data abstraction to the database [171], it still performs considerably to the native API within the cluster. If the model is applied in specific data environments with tailored APIs, there would not be a significant impact on the database's performance by integrating a third-party API. Astronomy and space science, healthcare informatics, and business intelligence typically employ multiple applications for specific purposes. These environments already have tailored APIs that have been in use over time and expect compatibility with new database

applications without necessitating changes to the existing infrastructure or affecting the overall database performance. It is crucial for our model to be effective in these environments without introducing new tools and methodologies to environments that already utilize a myriad of tools.

However, to a certain extent, we encourage the adoption of new tools to align with the design principle of 'using the right tools for a specific task. If these new tools prove to be more effective for a particular task than the existing ones, there is a compelling reason to embrace them. They often bring additional features that can simplify the storage challenge. For instance, scalable HDF5 files and scientific databases in astronomy. While both solutions are capable of storing vast amounts of scientific data, databases offer supplementary features such as indexing, query optimization, and efficient file organization that may be lacking in conventional files. Overall, our results will guide scientists, researchers, and collaborators from regions other than MeerKAT/SKA in South Africa, who would connect to the RFI database via a third-party API rather than the native API. This is useful to a multi-disciplinary environment at SKA, which interfaces engineers, astronomers, and software engineers who require several APIs to tailor their work.

Furthermore, the results suggest that the Join queries are costly to execute compared to simple, complex, and aggregate queries, irrespective of the API environment. It is difficult for a Join query to fetch data from several stores distributed in different locations, yet common in our RFI database cluster. Similarly, the performance of a cross-island query is shown to be affected by the distinct storage, as each acts as a physical barrier when data is accessed across storages. During access, data from one storage needs to be translated into a local island's data model before it is processed and shared with other islands. Eventually, all these tasks, including identifying the join attribute, matching the join tuples, and displaying the matched results, affect the execution time, which, in turn, affects the overall database performance. This is one of the challenges of querying across heterogeneous databases [215, 194]. Gadepally et al [131], suggest that more work needs to be done in optimizing query processing in a polystore system to scale more efficiently.

Our results have shown that storing SciDB data, particularly array-value data, in PSQL improves response times, especially under lower user loads. However, it is expected that PSQL's performance might decline when dealing with a high number of users requesting bulk data. For the MeerKAT and SKA international community, storing array data in SciDB would be highly beneficial during their large survey observations, which require a higher level of scalability, particularly when scientists from around the globe work concurrently on massive astronomical array data sets. Conversely, storing Accumulo data within PSQL would affect PSQL's response times. Therefore, it is recommended to keep Accumulo key-value data in the Accumulo store. Accumulo data includes various types of data, such as images, documents, text, or notes, which could be challenging to store in relational tables (PSQL store). Furthermore, latencies are not affected when storing SciDB or Accumulo data in PSQL. These values remain below the maximum acceptable latency, thus not affecting the model's performance.

Considering that SKA International is still under construction, we anticipate the growth of astronomical projects and an increase in data demands. It would be beneficial for these projects to store data in its appropriate store without the need for data transformation at a later stage. The more data is transformed, the more it loses its quality [17, 20]. Therefore, our results suggest storing array data in SciDB, key-value data in Accumulo, and relational data in PSQL. Each of these stores

leverages a specific data structure. With the completion of the SKA telescope and the growing number of users, neither SciDB array data nor Accumulo data would thrive in PSQL. This suggestion underscores the core principle of this research, which is to store each data in its appropriate store.

Our results have shown the model to be effective when retrieving bulk data from other storage systems, such as Accumulo or SciDB, into PSQL. PSQL boasts advanced query processing capabilities applicable to various storage configurations, developed over time [105]. If applied to projects like CyberSKA [216], a global platform responsible for delivering data to astronomers worldwide, it would foster collaboration in large science projects that require moving massive image data from one data center to another. Applying the database model to such projects would not only facilitate bulk data transfers but also empower astronomers to promptly remove corrupted data by cross-matching any detected RFI with the known RFI data in the database. For an unknown RFI detected, the database should facilitate further investigation. Consequently, we ensure that the data being transferred is examined thoroughly, before it is distributed to researchers worldwide for analysis.

Scientists, researchers, and students associated with the IDIA [98] can employ the database model to address the RFI identification problem. Researchers could utilize the model as an input for signal characterization (a critical step in identifying RFI). Characterizing RFI signals enables astronomers to classify RFI by source, strength, geographical location, or source position [96, 69]. Moreover, further studies can be conducted to investigate how to plug machine learning models to enhance the capabilities of the database model. Previous studies have shown that machine learning algorithms such as K-Nearest Neighbors, Random Forest Classifier, and deep learning Convolutional Neural Networks can complement human visual inspection for RFI signals in radio astronomy [94, 95]. Similarly, advanced visualization techniques can be further researched to effectively represent complex RFI data. Visualization in radio astronomy plays a crucial role in providing a more comprehensive understanding of complex astronomical data, especially in the context of large survey observations [22, 217]. Therefore, when the database model is plugged with advanced visualization and classification machine learning algorithms, it empowers astronomers to discern intricate details and patterns within complex RFI signals. Consequently, it can generate accurate RFI predictions, making a significant contribution to the identification problem. This capability can also benefit other radio observatories, such as NRAO and Hartebeesthoek Radio Astronomy Observatory (HartRAO) [218], which are looking to localize their unique RF environments. It's worth noting that each radio observatory operates within its distinct RF environment, often influenced by varying geographical settings. Observatories situated near commercial regions frequently contend with a wide range of RFI transmissions from sources like telecommunication towers, car ignitions, and Wi-Fi, while those located farther away experience lower interference levels.

Lastly, our results are limited to 25600 records per query download. Similarly, limited are to a maximum number of 36 concurrent, each requesting 1 MB of data per download. The number of records (> 25600) or more users (> 36) would overwhelm the current setup, thus causing several timeouts with no results, which is associated with the limited specifications of the host machine (database server) of 25 GB RAM and 11 TB of disk capacity. We, therefore, suggest powerful host machines to increase the number of concurrent users and the capacity to handle more data records per query. Further, the study did not evaluate the impact of Workload E (Short ranges) and F (Read-Modify) on the YSCB benchmark. Workload

E constitutes only 5% of the workload, yet the focus is on bulk queries/ workloads (>50% Workload). On the other hand, Workload F represents 50% but is not supported across the three data stores of the RFI database model. Lastly, the polystore framework adopted in this study is complex, whose performance may depend on other components within the database cluster, such as API middleware, network stability, and choice of store. Despite the above components, the framework has shown promising results in scientific data.

In summary, the database model performs well with increasing data volumes, users, and varying workloads. The model exhibits low response times, and the latency figures in all our experiments remain well within the acceptable maximum of less than one second, indicating the strong performance of our model. Furthermore, the model demonstrates effectiveness primarily for Reads, followed by Updates and Inserts. Our model also enables quick queries across different storage systems. Lastly, it is efficient to store each data in its native structure as dictated by its inherent format, especially when considering scalability as a core requirement for any storage application.

# Chapter 7

# Conclusions

This work aimed to develop a scalable database model suitable for RFI storage at the MeerKAT radio telescope. Thus, the main contribution of this study lies in designing a scalable database model that stores and quickly retrieves RFI data. The RFI database provides integrated storage that enables efficient storage of RFI data in different formats. The database model also provides a basic interface (native API) that supports bulk uploads into the database. Lastly, this work addresses the storage gap in the RFI mitigation pipeline at the MeerKAT radio telescope in South Africa.

We found that the model performs well with increasing data volumes, users, and varying workloads, with a low response time of less than 12 seconds for a 1 MB query of RFI data. The latency is consistently low, typically less than 0.14 seconds, and consistently below the maximum limit of one second. Other findings show a similar performance impact between the native and a third-party API. Furthermore, the findings indicate that the database model performs well for frequent Reads, followed by Updates, and then Insert operations. Our model facilitates quick queries across different storage systems within a cluster and emphasizes the core design principle underpinning this work: storing each type of data in its appropriate storage system.

Since the focus of this research was on creating a scalable RFI storage for the MeerKAT/SKA radio telescope, RFI identification and a detailed analysis of the RFI signal were not covered. These two requirements are essential in the RFI mitigation pipeline. Therefore, further work is required to explore how best to plug detection algorithms into the RFI database to detect or identify new RFI automatically by crossing the unknown RFI with the database. Also, more work is needed to investigate how the RFI database could support the analysis of timely statistics without necessarily introducing an additional tool external to the database.

Future work could add to the native database API to support a variety of complex queries, as well as offer support for more structured querying, such as SQL, in traditional databases. If extended to suit more queries, additional experiments could be conducted to assess the impact of several workloads on the database (such as Workloads E and F discussed in our limitations).

Additionally, the database query output is formatted in plain text from the standard output (stdout) (see Appendix C). The output from large queries in plain text is problematic on display. Hence, more work is needed to improve the visualization of the RFI database, particularly for large survey queries. With such additions, the database model could become more than a database for scientists and be useful for visualization.

Our results may serve as thresholds for scalability considerations in database system design. However, further research is required to develop approaches for building a more scalable model, especially to accommodate larger user loads. For instance, new methods are needed to run an integrated database using parallel

servers. The addition of more servers would significantly reduce the overhead on the host machine. These enhancements to the current implementation imply that a larger user load could be considered to assess the model's impact, potentially leading to improvements in its performance.

Integrating data from two or more systems remains an ongoing challenge within the database community. Each system inherently differs from the others, characterized by distinct data models, languages, and storage engines. This diversity makes query processing and optimization a daunting task, necessitating a significant effort to develop custom integration algorithms. Data integration within a heterogeneous data environment is currently a focal point of research in the realm of large and diverse databases [16, 139]. Although there is no universal solution, we can provide some general guidelines to approach this complex issue.

Firstly, the initial step is often to extract the desired data from different database systems and convert it into a common format suitable for integration. This frequently involves performing data type conversions to ensure meaningful joins. Afterwards, custom code may be required to execute sorting, comparison, and merging operations, aligning with specific join conditions. Upon completing the sort-merge join operations, the resulting query output can serve as an intermediate result for further database operations or be saved as a separate file within a specific database system. It's essential to recognize that this entire process demands continuous updates as new data emerges in each of the databases within the cluster

In conclusion, as scientists collect large and diverse scientific data from large survey observations, it has become crucial that scalability be a primary consideration in the design of scientific databases. In particular, scientists need to become database-friendly and reduce the over-dependence on traditional file systems (such as CSV) for storage as they lack full database functionalities. This thesis designs and implements a scalable database model to store RFI data using a polystore framework. Lastly, this work contributes to the idea of fully developing the polystore framework into a standard recommended for scientific data storage.

*PERSONAL REMARKS* – revising this thesis has been an overwhelming journey, one that initially felt like an impossible challenge. However, to my surprise, it turned into the most profound and fulfilling moment of my life. Over this period, I did not merely engage with my thesis; I embraced it as if it were my own child. The affection and attachment that blossomed during this brief phase of revision exceeded the deep connection I had developed over the years of crafting this thesis. As I delved into the process of refining and perfecting every paragraph, my appreciation for my work deepened. It was as if I was discovering the layers of a masterpiece I didn't fully comprehend before.

I hold a profound gratitude for the examiner's wisdom in requesting these revisions. Through this relentless journey, I have come to believe that this version truly represents the epitome of my understanding in this field of study. I eagerly anticipate the opportunity to share my knowledge and expertise with the entire science and local communities.

Finally, the most significant lesson I have gained during this period can be captured competently by a well-known Japanese proverb: *"Fall seven times, stand up eight"*. It has become my personal mantra, a reminder that in life, resilience and the courage to rise once more are the genuine secrets to success.

# Bibliography


[1] Square Kilometre Array (SKA UK). *SKA Timeline*. 2021. URL: https://unitedkingdom.skatelescope.org/ska-project/ska-timeline/ (visited on 12/08/2021).

[2] F. Camilo et al. "Revival of the Magnetar PSR J1622–4950: Observations with MeerKAT, Parkes, *XMM-Newton* , *Swift* , *Chandra* , and *NuSTAR*". *The Astrophysical Journal* 856.2 (Apr. 2018), p. 180. ISSN: 1538-4357.

[3] J. A. Jonas et al. "The MeerKAT Radio Telescope". en. *Proceedings of MeerKAT Science: On the Pathway to the SKA — PoS(MeerKAT2016)*. Stellenbosch, South Africa: Sissa Medialab, Feb. 2018, p. 001. URL: https://pos.sissa.it/277/001.

[4] The African Data Intensive Research Cloud (ADIRC). *Towards the Sustainable Development Goals with the African Data Intensive Research Cloud*. 2017. URL: https://www.sarao.ac.za/wp-content/uploads/2017/07/ska_adirc_fact_sheet_2017.pdf (visited on 11/10/2017).

[5] R. D. Ekers and J. F. Bell. "Radio Frequency Interference". *Symposium-International Astronomical Union*. Vol. 199. Cambridge University Press. Feb. 2002, pp. 498–505.

[6] T. Chiwewe and G. Hancke. "A look at spectrum management policies for radio spectrum". English. *EngineerIT* 03 (Mar. 2015), pp. 47–49. ISSN: 1991-5047.

[7] Independent Communications Authority of South Africa (ICASA). *Regulating the Communications Sector in the Public Interest*. 2021. URL: https://www.icasa.org.za/pages/about-us-1 (visited on 04/08/2021).

[8] Medicina Radiotelescopes. *Interference Monitoring – The radioastronomical bands protection*. 2008. URL: https://www.med.ira.inaf.it/Interferenze_page_EN.htm (visited on 05/03/2021).

[9] J. M. Ford and K. D. Buch. "RFI mitigation techniques in radio astronomy". *2014 IEEE Geoscience and Remote Sensing Symposium*. Quebec City, QC, Canada: IEEE, July 2014, pp. 231–234.

[10] P. A. Fridman and W. A. Baan. "RFI mitigation methods in radio astronomy". *Astronomy & Astrophysics* 378.1 (Oct. 2001), pp. 327–344. ISSN: 0004-6361, 1432-0746. DOI: 10.1051/0004-6361:20011166.

[11] M. Kesteven. *The current status of RFI mitigation in radio astronomy*. Tech. rep. Tech. rep., Australia Telescope National Facility, 2009.

[12] The Independent Communications Authority of South Africa (ICASA). *National Radio Frequency Plan 2013*. Tech. rep. 36336. Republic of South Africa, 2013.

[13] R. P. Millenaar and A. J. Otto. "Innovations in instrumentation for RFI monitoring". *2016 Radio Frequency Interference (RFI)*. Socorro, NM, USA: IEEE, Oct. 2016, pp. 65–68.

[14] G. N. Balekaki, M. Kuttel, A. Schroeder, S. Blyth, and S. Berman. "A Scalable Database Model of RFI Data for the MeerKAT Radio Telescope". *Proceedings of the South African Institute of Computer Scientists and Information Technologists 2019*. SAICSIT '19. Skukuza, South Africa: Association for Computing Machinery, 2019, pp. 1–8.

[15] X. L. Dong and D. Srivastava. "Big data integration". *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. Brisbane, Australia: IEEE, Apr. 2013, pp. 1245–1248.

[16] P. Kumar. "An Overview of Architectures and Techniques for Integrated Data Systems Implementation". en. *Actionable Intelligence: Using Integrated Data Systems to Achieve a More Effective, Efficient, and Ethical Government*. Ed. by J. Fantuzzo and D. P. Culhane. New York: Palgrave Macmillan US, 2015, pp. 105–123. DOI: 10.1057/9781137475114_4.

[17] P. Buneman. *Why Scientists Don't Use Databases*. Apr. 2002. URL: https://studyres.com/doc/819747/why-scientists-don-t-use-databases (visited on 09/10/2023).

[18] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. "An Overview of the HDF5 Technology Suite and Its Applications". *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. AD '11. Uppsala, Sweden: Association for Computing Machinery, 2011, 36–47.

[19] R. Rew and G. Davis. "NetCDF: an interface for scientific data access". *IEEE Computer Graphics and Applications* 10.4 (July 1990), pp. 76–82. ISSN: 1558-1756. DOI: 10.1109/38.56302.

[20] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. "Scientific Data Management in the Coming Decade". *ACM Sigmod Record* 34.4 (Oct. 2005), pp. 34–41. DOI: 10.1145/1107499.1107503.

[21] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. "SciDB: A Database Management System for Applications with Complex Analytics". *Computing in Science & Engineering* 15.3 (May 2013), pp. 54–62. DOI: 10.1109/MCSE.2013.19.

[22] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. "A demonstration of SciDB: a science-oriented DBMS". en. *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1534–1537. DOI: 10.14778/1687553.1687584.

[23] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2020. URL: https://www.postgresql.org/ (visited on 11/12/2020).

[24] B. Han, Y. Zhang, S. Zhong, and Y. Zhao. "Astronomical data fusion tool based on PostgreSQL". eng. *Research in Astronomy and Astrophysics* 16.11 (Nov. 2016), p. 178. DOI: 10.1088/1674-4527/16/11/178.

[25] M. Planthaber G.. Stonebraker and F. James. "EarthDB: Scalable Analysis of MODIS Data Using SciDB". *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. BigSpatial '12. Redondo Beach, California: ACM, 2012, pp. 11–19.

[26] J. Kantor, T. Axelrod, J. Becla, K. Cook, S. Nikolaev, R. Gray J.and Plante, M. Nieto-Santisteban, A. Szalay, and A. Thakar. "Designing for peta-scale in the LSST database". *Astronomical Data Analysis Software and Systems XVI*. Vol. 376. Arizona, USA, Oct. 2007, p. 3.

[27] N. R. Tanvir, A. J. Levan, C. González-Fernández, O. Korobkin, I. Mandel, S. Rosswog, J. Hjorth, P. D'Avanzo, A. S. Fruchter, C. L. Fryer, T. Kangas, B. Milvang-Jensen, S. Rosetti, D. Steeghs, R. T. Wollaeger, Z. Cano, C. M. Copperwheat, S. Covino, V. D'Elia, A. De Ugarte Postigo, P. A. Evans, W. P. Even, S. Fairhurst, R. Figuera Jaimes, C. J. Fontes, Y. I. Fujii, J. P. U. Fynbo, B. P. Gompertz, J. Greiner, G. Hodosan, M. J. Irwin, P. Jakobsson, U. G. Jørgensen, D. A. Kann, J. D. Lyman, D. Malesani, R. G. McMahon, A. Melandri, P. T. O'Brien, J. P. Osborne, E. Palazzi, D. A. Perley, E. Pian, S. Piranomonte, M. Rabus, E. Rol, A. Rowlinson, S. Schulze, P. Sutton, C. C. Thöne, K. Ulaczyk, D. Watson, K. Wiersema, and R. A. M. J. Wijers. "The Emergence of a Lanthanide-rich Kilonova Following the Merger of Two Neutron Stars". *The Astrophysical Journal* 848.2 (Oct. 2017), p. L27. DOI: 10.3847/2041-8213/aa90b6.

[28] João Alves, Catherine Zucker, Alyssa A. Goodman, Joshua S. Speagle, Stefan Meingast, Thomas Robitaille, Douglas P. Finkbeiner, Edward F. Schlafly, and Gregory M. Green. "A Galactic-scale gas wave in the solar neighbourhood". en. *Nature* 578.7794 (Feb. 2020), pp. 237–239. DOI: 10.1038/s41586-019-1874-z.

[29] Keith W. Bannister, Jamie Stevens, Artem V. Tuntsov, Mark A. Walker, Simon Johnston, Cormac Reynolds, and Hayley Bignall. "Real-time detection of an extreme scattering event: Constraints on Galactic plasma lenses". en. *Science* 351.6271 (Jan. 2016), pp. 354–356. DOI: 10.1126/science.aac7673.

[30] C.L. Carilli and S. Rawlings. "Motivation, key science projects, standards and assumptions". en. *New Astronomy Reviews* 48.11-12 (Dec. 2004), pp. 979–984. DOI: 10.1016/j.newar.2004.09.001.

[31] R. J. H. Dunn, R. P. Fender, E. G. Körding, T. Belloni, and C. Cabanac. "A global spectral study of black hole X-ray binaries". en. *Monthly Notices of the Royal Astronomical Society* 403.1 (Mar. 2010), pp. 61–82. DOI: 10.1111/j.1365-2966.2010.16114.x.

[32] T. Kahn. *Telescope project makes progress on radio-quiet zone*. en-ZA. Apr. 2017. URL: https://www.businesslive.co.za/bd/national/science-and-environment/2017-04-18-telescope-project-makes-progress-on-radio-quiet-zone/ (visited on 12/10/2021).

[33] B. Weeden. "Radio Frequency Spectrum, Interference and Satellites Fact Sheet". *Secure World Foundation* 25 (2013), pp. 1–4.

[34] K. G Jansky. "Electrical disturbances of apparently extraterrestrial origin". *Proceedings of the Institute of Radio Engineers* 21.10 (1933), pp. 1387–1398.

[35] G. Reber. "Notes: cosmic static". *Astrophysical Journal* 91.621 (1940), p. 1982.

[36] J. Delhaize, I. Heywood, M. Prescott, M. J. Jarvis, I. Delvecchio, I. H. Whittam, S. V. White, M. J. Hardcastle, C. L. Hale, J. Afonso, et al. "MIGHTEE: Are giant radio galaxies more common than we thought?" *Monthly Notices of the Royal Astronomical Society* 501.3 (Jan. 2021), pp. 3833–3845. DOI: 10.1093/mnras/staa3837.

[37] D. Li, R. Nan, Z. Pan, C. Jin, L. Zhu, Q. Wang, P. Jiang, K. Xu, C. Li, and C. Li. "The Five-hundred-meter Aperture Spherical radio Telescope (FAST) project". *2015 International Topical Meeting on Microwave Photonics (MWP)*. Paphos, Cyprus: IEEE, Oct. 2015, pp. 1–3.

[38] Z. Haiyan, R. Nan, P. Bo, X. Yuebing, C. Jin, J. Li, Z. Xiaonian, and G. Long. "Proposed radio quiet zone around FAST in China". *2013 Asia-Pacific Symposium on Electromagnetic Compatibility (APEMC)*. Melbourne, Australia: IEEE, May 2013, pp. 1–3.

[39] R. M. Prestage, K. T. Constantikes, T. R. Hunter, L. J. King, R. J. Lacasse, F. J. Lockman, and R. D. Norrod. "The green bank telescope". *Proceedings of the IEEE* 97.8 (Aug. 2009), pp. 1382–1390. DOI: 10.1109/JPROC.2009.2015467.

[40] R. D. Norrod, S. Srikanth, and M. Balister. "Receiver and optics designs for the 100-meter Green Bank telescope". *1992 IEEE Microwave Symposium Digest MTT-S*. Albuquerque, NM, USA: IEEE, June 1992, pp. 1365–1368.

[41] A. Richard Thompson, James M. Moran, and George W. Swenson. *Interferometry and Synthesis in Radio Astronomy*. 3rd ed. Astronomy and Astrophysics Library. Cham, Switzerland: Springer International Publishing, 2017. ISBN: 9783319444291 9783319444314.

[42] W. D. Cotton, K. Thorat, J. J. Condon, B. S. Frank, G. Józsa, S. V. White, R. Deane, N. Oozeer, M. Atemkeng, L. Bester, et al. "Hydrodynamical backflow in X-shaped radio galaxy PKS 2014- 55". *Monthly Notices of the Royal Astronomical Society* 495.1 (June 2020), pp. 1271–1283. DOI: 10.1093/mnras/staa1240.

[43] A. Hellerschmied, L. Plank, A. Neidhardt, R. Haas, J. Böhm, C. Plötz, and J. Kodet. "Observing Satellites with VLBI Radio Telescopes". *International VLBI Service for Geodesy and Astrometry 2014 General Meeting Proceedings:" VGOS: The New VLBI Network*. Shanghai, China, Aug. 2014, pp. 441–445.

[44] A. R. Thompson. "Fundamentals of radio interferometry". *Synthesis Imaging in Radio Astronomy II* 180 (1999), pp. 11–36.

[45] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, T. Lazio, and Joseph L. W. "The Square Kilometre Array". *Proceedings of the IEEE* 97.8 (Aug. 2009), pp. 1482–1496. DOI: 10.1109/JPROC.2009.2021005.

[46] C. Carilli and S. Rawlings. "Science with the Square Kilometer Array: Motivation, Key Science Projects, Standards and Assumptions". *New Astronomy Reviews* 48.11-12 (Dec. 2004), pp. 979–984. DOI: 10.1016/j.newar.2004.09.001.

[47] M. P. Rupen. "The Very Large Array expansion project". *Large Ground-based Telescopes*. Ed. by Jacobus M. O. and Larry M. S. Vol. 4837. International Society for Optics and Photonics. SPIE, 2003, pp. 119–128. DOI: 10.1117/12.457991.

[48] J. A. Hogbom and W. N. Brouw. "The synthesis radio telescope at Westerbork. Principles of operation, performance and data reduction". *Astronomy and Astrophysics* 33 (1974), p. 289.

[49] W. E. Wilson, R. H. Ferris, P. Axtens, A. Brown, E. Davis, G. Hampson, M. Leach, P. Roberts, S. Saunders, B. S. Koribalski, et al. "The Australia Telescope Compact Array broad-band backend: Description and first results". *Monthly Notices of the Royal Astronomical Society* 416.2 (2011), pp. 832–856.

[50] A. E. Schinckel, J. D. Bunton, T. J. Cornwell, I. Feain, and S. G. Hay. "The Australian SKA pathfinder". *Ground-based and Airborne Telescopes IV*. Ed. by Larry M. S., Roberto G., and Helen J. H. Vol. 8444. International Society for Optics and Photonics. SPIE, 2012, 84442A. DOI: 10.1117/12.926959.

[51] R. D. Ekers. "Square kilometre array (SKA)". *The Proceedings of the IAU 8th Asian-Pacific Regional Meeting, Volume 1*. Vol. 289. Epping, Australia, 2003, pp. 21–28.

[52] David R. DeBoer, Aaron R. Parsons, James E. Aguirre, Paul Alexander, Zaki S. Ali, Adam P. Beardsley, Gianni Bernardi, Judd D. Bowman, Richard F. Bradley, Chris L. Carilli, Carina Cheng, Eloy De Lera Acedo, Joshua S. Dillon, Aaron Ewall-Wice, Gcobisa Fadana, Nicolas Fagnoni, Randall Fritz, Steve R. Furlanetto, Brian Glendenning, Bradley Greig, Jasper Grobbelaar, Bryna J. Hazelton, Jacqueline N. Hewitt, Jack Hickish, Daniel C. Jacobs, Austin Julius, MacCalvin Kariseb, Saul A. Kohn, Telalo Lekalake, Adrian Liu, Anita Loots, David MacMahon, Lourence Malan, Cresshim Malgas, Matthys Maree, Zachary Martinot, Nathan Mathison, Eunice Matsetela, Andrei Mesinger, Miguel F. Morales, Abraham R. Neben, Nipanjana Patra, Samantha Pieterse, Jonathan C. Pober, Nima Razavi-Ghods, Jon Ringuette, James Robnett, Kathryn Rosie, Raddwine Sell, Craig Smith, Angelo Syce, Max Tegmark, Nithyanandan Thyagarajan, Peter K. G. Williams, and Haoxuan Zheng. "Hydrogen Epoch of Reionization Array (HERA)". *Publications of the Astronomical Society of the Pacific* 129.974 (Apr. 2017), p. 045001. ISSN: 0004-6280, 1538-3873. DOI: 10.1088/1538-3873/129/974/045001. URL: https://iopscience.iop.org/article/10.1088/1538-3873/129/974/045001 (visited on 06/16/2023).

[53] D. B. Davidson. "MeerKAT and SKA phase 1". *International Symposium on Antennas, Propagation and EM Theory (ISAPE2012)*. Xi'an, China, Oct. 2012, pp. 1279–1282.

[54] Zara Abdurashidova, James E. Aguirre, Paul Alexander, Zaki S. Ali, Yanga Balfour, Adam P. Beardsley, Gianni Bernardi, Tashalee S. Billings, Judd D. Bowman, Richard F. Bradley, Philip Bull, Jacob Burba, Steve Carey, Chris L. Carilli, Carina Cheng, David R. DeBoer, Matt Dexter, Eloy De Lera Acedo, Taylor Dibblee-Barkman, Joshua S. Dillon, John Ely, Aaron Ewall-Wice, Nicolas Fagnoni, Randall Fritz, Steven R. Furlanetto, Kingsley Gale-Sides, Brian Glendenning, Deepthi Gorthi, Bradley Greig, Jasper Grobbelaar, Ziyaad Halday, Bryna J. Hazelton, Jacqueline N. Hewitt, Jack Hickish, Daniel C. Jacobs, Austin Julius, Nicholas S. Kern, Joshua Kerrigan, Piyanat Kittiwisit, Saul A. Kohn, Matthew Kolopanis, Adam Lanman, Paul La Plante, Telalo Lekalake, David Lewis, Adrian Liu, David MacMahon, Lourence Malan, Cresshim Malgas, Matthys Maree, Zachary E. Martinot, Eunice Matsetela, Andrei Mesinger, Mathakane Molewa, Miguel F. Morales, Tshegofalang Mosiane, Steven G. Murray, Abraham R. Neben, Bojan Nikolic, Chuneeta D. Nunhokee, Aaron R. Parsons, Nipanjana Patra, Robert Pascua, Samantha Pieterse, Jonathan C. Pober, Nima Razavi-Ghods, Jon Ringuette, James Robnett, Kathryn Rosie, Peter Sims, Saurabh Singh, Craig Smith, Angelo Syce, Nithyanandan Thyagarajan, Peter K. G. Williams, and Haoxuan Zheng. "First Results from HERA Phase I: Upper Limits on the Epoch of Reionization 21 cm Power Spectrum". *The Astrophysical Journal* 925.2 (Feb. 2022), p. 221. ISSN: 0004-637X, 1538-4357. DOI: 10.3847/1538-4357/

ac1c78. URL: https://iopscience.iop.org/article/10.3847/1538-4357/ac1c78 (visited on 06/16/2023).

[55] R. A. Perley. "The very large array expansion project". *Radio Telescopes*. Vol. 4015. International Society for Optics and Photonics. 2000, pp. 2–7. DOI: 10.1117/12.926959.

[56] Square Kilometre Array Organisation (SKAO). *Software and Computing*. en. 2022. URL: https://www.skatelescope.org/software-and-computing/ (visited on 02/14/2022).

[57] T. J. Cornwell. "SKA and EVLA Computing Costs for Wide Field Imaging". en. *The Square Kilometre Array: An Engineering Perspective*. Ed. by Peter J. Hall. Dordrecht, Netherlands: Springer Netherlands, 2005, pp. 329–343. ISBN: 9781402037986. DOI: 10.1007/1-4020-3798-8_31.

[58] C. Fanti. "Radio telescope antennas: from single dish to multielement interferometer". *Proceedings of First MCCT-SKADS Training School — PoS(MCCT-SKADS)*. Medicina, Bologna Italy: Sissa Medialab, Aug. 2008, p. 1.

[59] A. Isella. *Interferometry Basics*. Mar. 2011. URL: https://slidetodoc.com/interferometry-basics-andrea-isella-caltech-alma-community-day/ (visited on 12/28/2021).

[60] M. T. Atemkeng, O. M. Smirnov, C. Tasse, G. Foster, and J. Jonas. "Using baseline-dependent window functions for data compression and field-of-interest shaping in radio interferometry". en. *Monthly Notices of the Royal Astronomical Society* 462.3 (Nov. 2016), pp. 2542–2558. DOI: 10.1093/mnras/stw1656.

[61] AR Thompson, GB Taylor, CL Carilli, and RA Perley. "Synthesis Imaging in Radio Astronomy II". ASP Conf. Ser. 180 (1999).

[62] S. A. A. Gillani. "RF Interference Monitoring for the Onsala Space Observatory". M.S. thesis. Göteborg, Sweden: Chalmers University of Technology, 2010.

[63] A. Boonstra. "Radio Frequency Interference Mitigation in Radio Astronomy". Ph.D. thesis. The Netherlands: Delft University of Technology, 2005.

[64] Square Kilometre Array Expert Panel on Radio Frequency Interference. *A report on the strengths and weaknesses of the currrent radio frequency interference environment as measured at the SKA candidate sites*. Tech. rep. Square Kilometre Array, Nov. 2011.

[65] Mike D. E. Turley, Andrew J. Heitmann, and Robert S. Gardiner-Garden. "Ionogram RFI Rejection Using an Autoregressive Interpolation Process". *Radio Science* 54.1 (Jan. 2019), pp. 135–150. DOI: 10.1029/2018RS006683.

[66] Thomas L. Wilson, Kristen Rohlfs, and Susanne Hüttemeister. *Tools of Radio Astronomy*. 6th ed. Astronomy and Astrophysics Library. Berlin, Germany: Springer, 2013. ISBN: 978-3-642-39949-7.

[67] C. Merwe. "Culprit and victim management RFI environment for a radio astronomy site". M.S. thesis. Stellenbosch, South Africa: Stellenbosch University, South Africa, 2012.

[68] Philippa Hillebrand. *Detection and Visualisation of Radio Frequency Interference*. Project report. Cape Town, South Africa: University of Cape Town, South Africa, 2014.

[69] D. Czech, A. K. Mishra, and M. Inggs. "Time domain classification of transient radio frequency interference". *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. Beijing, China: IEEE, July 2016, pp. 302–305. DOI: 10.1109/IGARSS.2016.7729071.

[70] J. G. Porko. "Radio frequency interference in radio astronomy". M.S. thesis. Espoo, Finland: Aalto University School of Electrical Engineering, 2011.

[71] A. E. E. Rogers, P. Pratap, J. C. Carter, and M. A. Diaz. "Radio frequency interference shielding and mitigation techniques for a sensitive search for the 327 MHz line of deuterium". *Radio science* 40.05 (Oct. 2005), pp. 1–10. DOI: 10.1029/2004RS003157.

[72] B. W. Stappers, J. W. T. Hessels, A. Alexov, K. Anderson, T. Coenen, T. Hassall, A. Karastergiou, V. Kondratiev, M. Kramer, J. Van Leeuwen, et al. "Observing pulsars and fast transients with LOFAR". *Astronomy & astrophysics* 530 (2011), A80. DOI: 10.1051/0004-6361/201116681.

[73] E. Petroff, J. W. T. Hessels, and D. R. Lorimer. "Fast radio bursts". *The Astronomy and Astrophysics Review* 27.1 (Dec. 2019), p. 4. DOI: 10.1007/s00159-019-0116-6.

[74] S. A. Tyul'bashev, V. S. Tyul'bashev, and V. M. Malofeev. "Detection of 25 new rotating radio transients at 111 MHz". *Astronomy & Astrophysics* 618 (Oct. 2018), A70. DOI: 10.1051/0004-6361/201833102.

[75] L. Driessen. "A new era of radio transients". en. *Astronomy & Geophysics* 61.5 (Oct. 2020), pp. 5.12–5.17. DOI: 10.1093/astrogeo/ataa068.

[76] D. Czech, A. K. Mishra, and M. Inggs. "Distinguishing between pulsars and transient RFI in the time domain". *2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. Fort Worth, Texas, USA, 2017, pp. 1236–1239.

[77] T. An, X. Chen, P. Mohan, and B. Lao. "Radio Frequency Interference Mitigation". *arXiv:1711.01978 [astro-ph]* (Nov. 2017). arXiv: 1711.01978. DOI: 10.15940/j.cnki.0001-5245.2017.05.002.

[78] A. R. Offringa, R. B. Wayth, N. Hurley-Walker, D. L. Kaplan, N. Barry, A. P. Beardsley, M. E. Bell, G. Bernardi, J. D. Bowman, F. Briggs, et al. "The low-frequency environment of the Murchison Widefield Array: radio-frequency interference analysis and mitigation". *Publications of the Astronomical Society of Australia* 32 (Jan. 2015). DOI: 10.1017/pasa.2015.7.

[79] South African Radio Astronomy Observatory (SARAO). *South African Radio Astronomy Observatory (SARAO)*. 2023. URL: https://www.sarao.ac.za/ (visited on 11/05/2023).

[80] P. C. Crane and L. A. Hillenbrand. "Estimating Harmful Levels of Radio-Frequency Radiation". en. *International Astronomical Union Colloquium* 112 (1991), pp. 258–266. ISSN: 0252-9211. DOI: 10.1017/S0252921100004085.

[81] SKA Expert Panel. *Expert Panel on Radio Quiet Zone and RFI Regulation Report*. Tech. rep. Square Kilometre Array, Oct. 2011.

[82] D. B. Davidson. "The SKA and the MeerKAT precursor – Extreme antenna engineering". *The 8th European Conference on Antennas and Propagation (EuCAP 2014)*. The Hague, Netherlands: IEEE, Apr. 2014, pp. 1216–1219.

[83] South African Radio Astronomy Observatory (SARAO). *MeerKAT radio telescope inaugurated in South Africa – reveals clearest view yet of center of the Milky Way – SARAO*. 2018. URL: https://www.sarao.ac.za/media-releases/meerkat-radio-telescope-inaugurated-in-south-africa-reveals-clearest-view-yet-of-center-of-the-milky-way/ (visited on 12/12/2018).

[84] The SKA telescope. *The Baseline Design of SKA1*. 2018. URL: https://astronomers.skatelescope.org/ska/ (visited on 12/12/2018).

[85] S. Wild. "Powering the world's largest telescope". en. *Physics Today* (Jan. 2019). DOI: 10.1063/PT.6.2.20190131a.

[86] Square Kilometre Array Organisation (SKAO). *The SKA Project*. en. 2022. URL: https://www.skatelescope.org/the-ska-project/ (visited on 02/15/2022).

[87] A. McPherson et al. "Report and Options for Re-baselining of SKA-1". *SKAO, SKA-TEL-SKO-0000229, rev1* (Apr. 2015), pp. 04–03.

[88] South African Astronomical Observatory (SARAO). *MeerKAT radio telescope*. 2020. URL: https://www.sarao.ac.za/gallery/meerkat/ (visited on 09/02/2020).

[89] M. Ramatsoku, M. Murgia, V. Vacca, P. Serra, S. Makhathini, F. Govoni, O. Smirnov, LAL. Andati, E. de Blok, G. Józsa, et al. "Collimated synchrotron threads linking the radio lobes of ESO 137-006". *Astronomy & Astrophysics* 636 (Apr. 2020), p. L1. DOI: 10.1051/0004-6361/202037800.

[90] South African Government. *Astronomy Geographic Advantage Act 21 of 2007*. 2007. URL: https://www.gov.za/documents/astronomy-geographic-advantage-act (visited on 12/10/2021).

[91] W. A. Baan. "RFI mitigation in radio astronomy". *2011 XXXth URSI General Assembly and Scientific Symposium*. Istanbul, Turkey: IEEE, Aug. 2011, pp. 1–2.

[92] J. Raza, A.-J. Boonstra, and A.J. van der Veen. "Spatial filtering of RF interference in radio astronomy". *IEEE Signal Processing Letters* 9.2 (Feb. 2002), pp. 64–67. DOI: 10.1109/97.991140.

[93] P. A. Fridman. "RFI excision using a higher order statistics analysis of the power spectrum". *Astronomy & Astrophysics* 368.1 (Mar. 2001), pp. 369–376. DOI: 10.1051/0004-6361:20000552.

[94] Stephen Itschner and Xin Li. "Radio Frequency Interference (RFI) Detection in Instrumentation Radar Systems: a Deep Learning Approach". *2019 IEEE Radar Conference (RadarConf)*. Boston, MA, USA: IEEE, Apr. 2019, pp. 1–5. DOI: 10.1109/RADAR.2019.8835604.

[95] Olorato Mosiane, Nadeem Oozeer, Arun Aniyan, and Bruce A. Bassett. "Radio Frequency Interference Detection using Machine Learning." *IOP Conference Series: Materials Science and Engineering* 198 (May 2017), p. 012012. DOI: 10.1088/1757-899X/198/1/012012.

[96] C. Schollar. "RFI Monitoring for the MeerKAT Radio Telescope". M.S. thesis. Cape Town, South Africa: University of Cape Town, South Africa, 2015.

[97] National Integrated Cyber Infrastructure System (NICIS). *Advancing High-Performance Computing in South Africa: The CHPC – NICIS*. en-US. 2022. URL: https://www.nicis.ac.za/chpc/ (visited on 09/27/2022).

[98] Inter-University Institute for Data Intensive Astronomy (IDIA). *IDIA – from big data to big ideas*. en-ZA. 2022. URL: https://www.idia.ac.za/ (visited on 09/27/2022).

[99] D. Aikema, B. Frank, R. Simmonds, Y. Grange, S. Sánchez-Expósito, S. Gaudet, and S. Goliath. "Data Delivery Architecture for MeerKAT and the SKA". *Astronomical Data Analysis Software and Systems XXVII* 522 (Apr. 2020), p. 331.

[100] South African Radio Astronomy Observatory (SARAO). *Breakthrough Listen to incorporate the MeerKAT array in its existing search for extraterrestrial signals and technosignatures*. en-US. 2022. URL: https://www.sarao.ac.za/media-releases/breakthrough-listen-to-incorporate-the-meerkat-array-in-its-existing-search-for-extraterrestrial-signals-and-technosignatures/ (visited on 09/27/2022).

[101] P. Thavasimani and A. Scaife. "Square Kilometre Array : Processing Voluminous MeerKAT Data on IRIS" (May 2021). arXiv:2105.14613 [astro-ph]. DOI: 10.48550/ARXIV.2105.14613.

[102] S. Rautmare and D. M. Bhalerao. "MySQL and NoSQL database comparison for IoT application". *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*. Coimbatore, India., Oct. 2016, pp. 235–238.

[103] R. Cattell. "Scalable SQL and NoSQL data stores". *ACM SIGMOD Record* 39.4 (May 2011), pp. 12–27. DOI: 10.1145/1978915.1978919.

[104] R. Zafar, E. Yafi, M. F. Zuhairi, and H. Dao. "Big Data: The NoSQL and RDBMS review". *2016 International Conference on Information and Communication Technology (ICICTM)*. Bandung, Indonesia, May 2016, pp. 120–126.

[105] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". en. *Communications of the ACM* 13.6 (June 1970), pp. 377–387. DOI: 10.1145/362384.362685.

[106] N. Banothu, S. Bhukya, and K. V. Sharma. "Big-data: Acid versus base for database transactions". *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*. IEEE. Chennai, India, Mar. 2016, pp. 3704–3709.

[107] W. M. Chen, Y. T. Chen, P. C. Hsiu, and T. W. Kuo. "Multiversion Concurrency Control on Intermittent Systems". *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Westminster, Colorado, USA, Nov. 2019, pp. 1–8.

[108] Shojiro Muro, Tiko Kameda, and Toshimi Minoura. "Multi-version concurrency control scheme for a database system". en. *Journal of Computer and System Sciences* 29.2 (Oct. 1984), pp. 207–224. DOI: 10.1016/0022-0000(84)90031-X.

[109] D. G. Chandra. "BASE analysis of NoSQL database". *Future Generation Computer Systems* 52 (Nov. 2015), pp. 13–21. DOI: 10.1016/j.future.2015.05.003.

[110] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. English. 1st edition. Upper Saddle River, NJ, USA: Addison-Wesley Professional, Aug. 2012. ISBN: 9780321826626.

[111] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. "Database design for NoSQL systems". *International Conference on Conceptual Modeling*. Springer. Atlanta, GA, USA, Oct. 2014, pp. 223–231.

[112] Y. Huang and T. Luo. "NoSQL Database: A Scalable, Availability, High Performance Storage for Big Data". *Pervasive Computing and the Networked World*. Vina del Mar, Chile: Springer International Publishing, Dec. 2014, pp. 172–183.

[113] C. Strauch and W. Kriha. "NoSQL databases". *Lecture Notes, Stuttgart Media University* 20 (2011), p. 24.

[114] J. Kepner, V. Gadepally, D. Hutchison, H. Jananthan, T. Mattson, S. Samsi, and A. Reuther. "Associative array model of SQL, NoSQL, and NewSQL databases". *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA, Sept. 2016, pp. 1–9.

[115] N. Askitis. "Fast and Compact Hash Tables for Integer Keys". *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*. ACSC '09. Wellington, New Zealand: Australian Computer Society, Inc., Jan. 2009, 113–122.

[116] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. DOI: 10.1145/1365815.1365816.

[117] A. Cordova, B. Rinaldi, and M. Wall. *Accumulo Application Development, Tables Designs, and Best Practices*. Ed. by M. Beaugureau. 1st Edition. O'Reilly Media, Inc, 2015.

[118] The Apache Software Foundation. *Manage massive amounts of data, fast, without losing sleep*. 2021. URL: https://cassandra.apache.org/ (visited on 02/03/2021).

[119] V. Bhupathiraju and R. P. Ravuri. "The dawn of Big Data - Hbase". *2014 Conference on IT in Business, Industry and Government (CSIBIG)*. Indore, India, Mar. 2014, pp. 1–4.

[120] D. Borthakur. *HDFS Architecture Guide*. 2021. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (visited on 11/01/2021).

[121] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Reuther, A. Rosa, and C. Yee. "Achieving 100,000,000 database inserts per second using Accumulo and D4M". *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA, Sept. 2014, pp. 1–6.

[122] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. "Foundations of JSON schema". *Proceedings of the 25th International Conference on World Wide Web*. Montreal, Canada, Apr. 2016, pp. 263–273.

[123] C. Asaad and K. Baïna. "NoSQL Databases–Seek for a Design Methodology". *International Conference on Model and Data Engineering*. Springer International Publishing. Cham, Sept. 2018, pp. 25–40.

[124] A. Pore. *NoSQL Data Architecture and Data Governance: Everything You Need to Know*. 2020. URL: https://www.dataversity.net/nosql-data-architecture-data-governance-everything-need-know/ (visited on 02/16/2020).

[125] MongoDB, Inc. *The database for modern applications*. 2020. URL: https://www.mongodb.com/ (visited on 11/14/2020).

[126] Neo4j, Inc. *The Native Graph Database for Today's Connected Applications*. 2020. URL: https://neo4j.com/neo4j-graph-database/ (visited on 11/12/2020).

[127] I. Katsov. *NOSQL data modeling techniques*. 2021. URL: https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/ (visited on 02/02/2021).

[128] T. Li, Y. Liu, Y. Tian, S. Shen, and W. Mao. "A Storage Solution for Massive IoT Data Based on NoSQL". *2012 IEEE International Conference on Green Computing and Communications*. Besancon, France, Nov. 2012, pp. 50–57.

[129] Michael Stonebraker and Lawrence A. Rowe. "The Design of POSTGRES". *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. SIGMOD '86. Washington, D.C., USA: Association for Computing Machinery, 1986, 340–355. ISBN: 0897911911. DOI: 10.1145/16894.16888. URL: https://doi.org/10.1145/16894.16888.

[130] J. M. Duggan, A. J. Elmore, T. Kraska, S. Madden, T. Mattson, and M. Stonebraker. "The BigDawg Architecture and Reference Implementation". *Eighth Annual New England Database Day*. Cambridge, Massachusetts, USA, Jan. 2016.

[131] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. "The BigDAWG polystore system and architecture". *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA, Sept. 2016, pp. 1–6.

[132] G. N. Balekaki, M. Kuttel, A. Schroeder, S. Blyth, and S. Berman. "Performance evaluation of an integrated RFI database for the MeerKAT/SKA radio telescope". *Conference of the South African Institute of Computer Scientists and Information Technologists 2020*. SAICSIT '20. Cape Town, South Africa: Association for Computing Machinery, Sept. 2020, pp. 29–34.

[133] Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, and Jeff Parkhurst. "Demonstrating the BigDAWG Polystore. System for Ocean Metagenomic Analysis". *CIDR*. 2017.

[134] K. Srivastava and N. Shekokar. "A Polyglot Persistence approach for E-Commerce business model". *2016 International Conference on Information Science (ICIS)*. Dublin, Ireland, Aug. 2016, pp. 7–11.

[135] S. Prasad and S. B. Avinash. "Application of polyglot persistence to enhance performance of the energy data management systems". *2014 International Conference on Advances in Electronics Computers and Communications*. Bangalore, India, Oct. 2014, pp. 1–6.

[136] P. Chen, V. Gadepally, and M. Stonebraker. "The BigDawg monitoring framework". *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2016, pp. 1–6. DOI: 10.1109/HPEC.2016.7761642.

[137] C. Jie, H. Wen, and C. Tingyou. "Research of Heterogeneous Database Integration system based on E-business". *2008 IEEE International Conference on Service Operations and Logistics, and Informatics*. Vol. 1. Beijing, China, Oct. 2008, pp. 186–189.

[138] Laura M Haas, Eileen Tien Lin, and Mary A Roth. "Data integration through database federation". *IBM Systems Journal* 41.4 (2002), pp. 578–596.

[139] J. Wang, Z. Miao, Y. Zhang, and B. Zhou. "Querying Heterogeneous Relational Database using SPARQL". *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. Shanghai, China, June 2009, pp. 475–480.

[140] BigDAWG Developers. *BigDAWG Documentation, Release 0.1*. 2017. URL: `https://bigdawg.mit.edu/sites/default/files/images/BigDAWGv01.pdf` (visited on 12/28/2020).

[141] Université de Strasbourg. *VizieR*. 2020. URL: `https://vizier.u-strasbg.fr/index.gml` (visited on 11/14/2020).

[142] F. Genova. "Strasbourg Astronomical Data Center (CDS)". en. *Data Science Journal* 12.0 (2013), WDS56–WDS60. ISSN: 1683-1470. DOI: `10.2481/dsj.WDS-007`. (Visited on 10/04/2022).

[143] M. Taylor. *TOPCAT: Tool for OPerations on Catalogues And Tables*. 2020. URL: `http://www.star.bris.ac.uk/~mbt/topcat/` (visited on 11/14/2020).

[144] Université de Strasbourg/CNRS. *SIMBAD Astronomical Database - CDS (Strasbourg)*. 2020. URL: `{http://simbad.u-strasbg.fr/simbad/}` (visited on 11/14/2020).

[145] M. Wenger, F. Ochsenbein, D. Egret, P. Dubois, F. Bonnarel, S. Borde, F. Genova, G. Jasniewicz, S. Laloë, S. Lesteven, et al. "The SIMBAD astronomical database-The CDS reference database for astronomical objects". *Astronomy and Astrophysics Supplement Series* 143.1 (Apr. 2000), pp. 9–22. DOI: `10.1051/aas:2000332`.

[146] P. G. Brown. "Overview of sciDB: large scale array storage, processing and analysis". en. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. Indianapolis Indiana USA: ACM, June 2010, pp. 963–968. ISBN: 9781450300322.

[147] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. "The architecture of SciDB". *International Conference on Scientific and Statistical Database Management*. Springer. Portland, OR, USA, July 2011, pp. 1–16.

[148] G. Lee Planthaber. "MODBASE: A SciDB-Powered System for Large-Scale Distributed Storage and Analysis of MODIS Earth Remote Sensing Data". M.S. thesis. Boston, MA, USA: Massachusetts Institute of Technology., 2012.

[149] Luboš Krčál and Shen-Shyang Ho. "A SciDB-based Framework for Efficient Satellite Data Storage and Query based on Dynamic Atmospheric Event Trajectory". *Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. Bellevue, WA, USA: ACM, Nov. 2015, pp. 7–14.

[150] J. Lee, D. J. Scott, M. Villarroel, G. D. Clifford, M. Saeed, and R. G. Mark. "Open-access MIMIC-II database for intensive care research". *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Boston, Massachusetts, USA, Aug. 2011, pp. 8315–8318.

[151] GEOTRACES. *An International Study of the Marine Biogeochemical Cycles of Trace Elements and Isotopes*. 2017. URL: `http://www.geotraces.org` (visited on 08/06/2017).

[152] A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, et al. "A demonstration of the bigdawg polystore system". *Proceedings of the VLDB Endowment* 8.12 (2015), p. 1908.

[153] J. Gray, ed. *The Benchmark handbook: for database and transaction processing systems*. eng. 2. ed., 2. [print.] The Morgan Kaufman series in data management systems. San Francisco, Calif: Morgan Kaufmann, 1994. ISBN: 9781558602922.

[154] D. J. DeWitt. "The Wisconsin Benchmark: Past, Present, and Future". *Computer Science Department, University of Winconsin* (1992), pp. 269–316.

[155] A. Dey, A. Fekete, R. Nambiar, and U. Röhm. "YCSB+T: Benchmarking web-scale transactional databases". *2014 IEEE 30th International Conference on Data Engineering Workshops*. Chicago, USA, Mar. 2014, pp. 223–230.

[156] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores". *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: Association for Computing Machinery, 2011.

[157] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB". *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 143–154.

[158] M. Thakur. "Benchmarking Top NoSQL Databases" (2022). URL: `https://www.academia.edu/26375336/Benchmarking_Top_NoSQL_Databases` (visited on 02/09/2022).

[159] I. Lungu, B. G. Tudorica, et al. "The development of a benchmark tool for nosql databases". *Database Systems Journal* 4.2 (2013), pp. 13–20.

[160] E. Tang and Y. Fan. "Performance comparison between five NoSQL databases". *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE. Taipa, Macau, China, Nov. 2016, pp. 105–109.

[161] V. Abramova, J. Bernardino, and P. Furtado. "Experimental evaluation of NoSQL databases". *International Journal of Database Management Systems* 6.3 (2014), p. 1.

[162] H. Matallah, G. Belalem, and K. Bouamrane. "Experimental comparative study of NoSQL databases: HBASE versus MongoDB by YCSB". *Comput. Syst. Sci. Eng* 32.4 (2017), pp. 307–317.

[163] S. H. Aboutorabi, M. Rezapour, M. Moradi, and N. Ghadiri. "Performance evaluation of SQL and MongoDB databases for big e-commerce data". *2015 International Symposium on Computer Science and Software Engineering (CSSE)*. Tabriz, Iran, Aug. 2015, pp. 1–7.

[164] H. Fatima and K. Wasnik. "Comparison of SQL, NoSQL and NewSQL databases for internet of things". *2016 IEEE Bombay Section Symposium (IBSS)*. Mumbai, India, Dec. 2016, pp. 1–6.

[165] H. Wadhwa K. & Pham. *Evaluating Data Latency for Real-Time Databases*. en. 2022. URL: `https://rockset.com/whitepapers/evaluating-data-latency-for-real-time-databases/` (visited on 02/01/2022).

[166] Atlassian. *Database latency impact on Jira performance | Jira | Atlassian Documentation*. 2022. URL: https://confluence.atlassian.com/jirakb/database-latency-impact-on-jira-performance-1082268864.html (visited on 02/01/2022).

[167] S. Barker, Y. Chi, Hyun J. Moon, H. Hacigümüş, and P. Shenoy. “"Cut me some slack": latency-aware live migration for databases”. *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT ’12. Berlin, Germany: Association for Computing Machinery, Mar. 2012, pp. 432–443.

[168] J. Wu. “Reducing the Tail Latency of a Distributed NoSQL Database”. M.S. thesis. Nebraska, USA: Dept. of Computer Science and Engineering, University of Nebraska - Lincoln, 2018.

[169] R. Elmasri and S. B. Navathe. “Fundamentals of database systems (6th Edition)”. Ed. by Michael Hirsch. Addison-Wesley, 2011. Chap. 10, p. 303.

[170] T. J Teorey. *Database modeling and design*. Morgan Kaufmann, 1999.

[171] C. Coronel and S. Morris. “Database Systems: Design, Implementation, and Management (12th Edition)”. Ed. by Cengage Learning. Cengage Learning, 2016. Chap. 3, p. 458.

[172] A. Hershey, G. Gardarin, and D. Reiner. “Database design: methodologies, tools, and environments (panel session)”. *ACM SIGMOD Record* 14.4 (1985), pp. 148–150.

[173] Google. *Collect and organize information big and small with Google Forms*. 2022. URL: https://www.google.com/forms/about/ (visited on 02/22/2020).

[174] C. Batini, S. Ceri, and S. Navathe. *Conceptual database design: an entity-relationship approach*. Redwood City, Calif: Benjamin/Cummings Pub. Co, 1992. ISBN: 9780805302448.

[175] T. A. Bruce. *Designing quality databases with IDEF1X information models*. New York, NY: Dorset House Publishing New York, 1992. ISBN: 9780932633187.

[176] P. J. Morris. “Relational database design and implementation for biodiversity informatics”. *PhyloInformatics* 7 (Nov. 2005), pp. 1–66.

[177] T. A. Halpin and A. J. Morgan. *Information modeling and relational databases*. 2nd ed. Morgan Kaufmann series in data management systems. Burlington, MA: Elsevier/Morgan Kaufman Publishers, 2008. ISBN: 9780123735683.

[178] E. Marcos, B. Vela, and J. M. Cavero. “A methodological approach for object-relational database design using UML”. *Software and Systems Modeling* 2.1 (Mar. 2003), pp. 59–72. DOI: 10.1007/s10270-002-0001-y.

[179] I. Puja, P. Poscic, and D. Jaksic. “Overview and comparison of several elational database modelling metodologies and notations”. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija, Croatia: IEEE, May 2019, pp. 1641–1646.

[180] K. Shin, C. Hwang, H. Jung, and H. Jung. “NoSQL database design using UML conceptual data model based on Peter Chen’s framework”. *International Journal of Applied Engineering Research* 12.5 (2017), pp. 632–636.

[181] G.T. Nguyen and D. Rieu. “Schema evolution in object-oriented database systems”. en. *Data & Knowledge Engineering* 4.1 (July 1989), pp. 43–67. DOI: 10.1016/0169-023X(89)90004-9.

[182] G. Moerkotte and A. Zachmann. "Towards more flexible schema management in object bases". *Proceedings of IEEE 9th International Conference on Data Engineering*. Vienna, Austria: IEEE Comput. Soc. Press, Apr. 1993, pp. 174–181.

[183] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. Seventh edition. New York, NY: McGraw-Hill, 2020. ISBN: 9780078022159 9781260515046.

[184] SciDB, Inc. *SciDB User's Guide*. 2013. URL: https://www.nersc.gov/assets/Uploads/scidb-userguide-12.3.pdf (visited on 12/28/2020).

[185] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser. "Performance Evaluation of NoSQL Databases: A Case Study". *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*. PABS '15. Austin, Texas, USA: Association for Computing Machinery, 2015, 5–10.

[186] Armando Fox and Eric A. Brewer. "Reducing WWW latency and bandwidth requirements by real-time distillation". en. *Computer Networks and ISDN Systems* 28.7-11 (May 1996), pp. 1445–1456. DOI: 10.1016/0169-7552(96)00027-X.

[187] M. Saraclar, M. Riley, E. Bocchieri, and V. Goffin. "Towards automatic closed captioning : low latency real time broadcast news transcription". en. *7th International Conference on Spoken Language Processing (ICSLP 2002)*. ISCA, Sept. 2002, pp. 1741–1744.

[188] Docker, Inc. *Empowering App Development for Developers | Docker*. en. 2021. URL: https://www.docker.com/ (visited on 10/17/2021).

[189] N. Naik. "Docker container-based big data processing system in multiple clouds for everyone". *2017 IEEE International Systems Engineering Symposium (ISSE)*. Vienna, Austria., Oct. 2017, pp. 1–7.

[190] B. Bashari Rad, H. Bhatti, and M. Ahmadi. "An Introduction to Docker and Analysis of its Performance". *IJCSNS International Journal of Computer Science and Network Security* 173 (Mar. 2017), p. 8.

[191] W. Velásquez, A. Munoz-Arcentales, and J. S. Rodriguez. "A Case Study: Ingestion Analysis of WSN Data in Databases using Docker". *2018 1st International Conference on Computer Applications Information Security (ICCAIS)*. Riyadh, Kingdom of Saudi Arabia, Apr. 2018, pp. 1–6.

[192] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. "The bigdawg polystore system". *ACM Sigmod Record* 44.2 (Aug. 2015), pp. 11–16. DOI: 10.1145/2814710.2814713.

[193] Sneha Mehta and Viral Mehta. "Hadoop ecosystem: an introduction". *International Journal of Science and Research (IJSR)* 5.6 (June 2016), pp. 557–562. DOI: 10.21275/v5i6.NOV164121. (Visited on 11/09/2023).

[194] V. Gadepally, K. OBrien, A. Dziedzic, A. Elmore, J. Kepner, S. Madden, T. Mattson, J. Rogers, Z. She, and M. Stonebraker. "Version 0.1 of the bigdawg polystore system". *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA, Sept. 2017, pp. 1–7.

[195] D. Stenberg. *Curl: command line tool and library for transferring data with URLs*. 2021. URL: https://curl.se/ (visited on 09/22/2021).

[196] B. Christudas. "cURL and Postman". *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Berkeley, CA: Apress, 2019, pp. 847–855. ISBN: 978-1-4842-4501-9.

[197] Roffit. *curl man page*. 2022. URL: https://www.mit.edu/afs.new/sipb/user/ssen/src/curl-7.11.1/docs/curl.html (visited on 01/24/2022).

[198] GNU GREP. *grep man page*. 2022. URL: https://linuxcommand.org/lc3_man_pages/grep1.html (visited on 02/01/2022).

[199] Jupyter. *JupyterLab: A Next-Generation Notebook Interface*. en. URL: https://jupyter.org (visited on 03/18/2022).

[200] Vazha Omanashvili. *JSON Generator – Tool for generating random data*. en. URL: https://json-generator.com (visited on 11/17/2023).

[201] M. Barata, J. Bernardino, and P. Furtado. "YCSB and TPC-H: Big Data and Decision Support Benchmarks". *2014 IEEE International Congress on Big Data*. Anchorage, AK, USA, 2014, pp. 800–801.

[202] Insomnia, Inc. *Design and debug APIs like a human, not a robot*. 2019. URL: https://insomnia.rest/ (visited on 07/10/2019).

[203] Postman, Inc. *Postman API Platform*. en. URL: https://www.postman.com/product/what-is-postman/ (visited on 12/12/2021).

[204] Inc MongoDB. *Data Model Design*. 2021. URL: https://docs.mongodb.com/manual/core/data-model-design/ (visited on 02/04/2021).

[205] R. Sen, A. Farris, and P. Guerra. *Benchmarking the Apache Accumulo Distributed Key–Value Store*. Tech. rep. June 2014. URL: https://accumulo.apache.org/papers/accumulo-benchmarking-2.1.pdf.

[206] O. Almootassem, S. H. Husain, D. Parthipan, and Q. H. Mahmoud. *A Cloud-based Service for Real-Time Performance Evaluation of NoSQL Databases*. arXiv:1705.08317 [cs]. May 2017. URL: http://arxiv.org/abs/1705.08317 (visited on 03/18/2022).

[207] J. Han, M. Song, and J. Song. "A Novel Solution of Distributed Memory NoSQL Database for Cloud Computing". *2011 10th IEEE/ACIS International Conference on Computer and Information Science*. Sanya, Hainan Island, China, 2011, pp. 351–355.

[208] Anindya Datta, Sarit Mukherjee, and Igor R. Viguier. "Buffer management in real-time active database systems". en. *Journal of Systems and Software* 42.3 (Sept. 1998), pp. 227–246. DOI: 10.1016/S0164-1212(98)10012-2. (Visited on 11/08/2023).

[209] Steve G. Langer. "Challenges for data storage in medical imaging research". en. *Journal of Digital Imaging* 24.2 (Apr. 2011), pp. 203–207. DOI: 10.1007/s10278-010-9311-8.

[210] N. Peek, J. H. Holmes, and J. Sun. "Technical challenges for big data in biomedicine and health: data sources, infrastructure, and analytics". en. *Yearbook of Medical Informatics* 23.01 (Aug. 2014), pp. 42–47. DOI: 10.15265/IY-2014-0018. (Visited on 11/03/2023).

[211] Anna Zvikhachevskaya, Garik Markarian, and Luydmila Mihaylova. "Quality of service consideration for the wireless telemedicine and e-health services". *2009 IEEE Wireless Communications and Networking Conference*. Budapest, Hungary: IEEE, Apr. 2009, pp. 1–6.

[212] Flávio RC Sousa, Leonardo O Moreira, Gustavo AC Santos, and Javam C Machado. "Quality of service for database in the cloud:" *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. Porto, Portugal: SciTePress - Science, and Technology Publications, Apr. 2012, pp. 595–601.

[213] N.R. Council, D.E.P. Sciences, B.P. Astronomy, C.R. Frequencies, and C.S.U.R. Spectrum. *Spectrum Management for Science in the 21st Century*. Washington, DC: National Academies Press, 2010. ISBN: 9780309151542.

[214] Roslan Umar, Z Abidin, Z Ibrahim, N Gasiprong, K Asanok, S Nammahachak, S Aukkaravittayapun, P Somboopon, A Prasit, N Prasert, et al. "The study of radio frequency interference (RFI) in Altitude effect on radio astronomy in Malaysia and Thailand". *Middle East Journal of Scientific Research* 14.6 (2013), pp. 861–866. DOI: 10.5829/idosi.mejsr.2013.14.6.2185. (Visited on 11/03/2023).

[215] S. Ceri, B. Pernici, and G. Wiederhold. "Distributed database design methodologies". *Proceedings of the IEEE* 75.5 (1987), pp. 533–546. DOI: 10.1109/PROC.1987.13771.

[216] Cameron Kiddle, A. R. Taylor, Jim Cordes, Olivier Eymere, Victoria Kaspi, Dan Pigat, Erik Rosolowsky, Ingrid Stairs, and A. G. Willis. "CyberSKA: An on-Line Collaborative Portal for Data-Intensive Radio Astronomy". *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*. GCE '11. Seattle, Washington, USA: Association for Computing Machinery, Nov. 2011, 65–72.

[217] 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 (July 2020), p. 100389. DOI: 10.1016/j.ascom.2020.100389. (Visited on 11/03/2023).

[218] Marisa Nickola, Jonathan Quick, and Ludwig Combrinck. *Hartebeesthoek Radio Astronomy Observatory (HartRAO)*. Tech. rep. HartRAO Network Station, Apr. 2013. URL: https://ivscc.gsfc.nasa.gov/publications/ar2013/nshartrao.pdf.

[219] IEEE Reach. *South Africa Spectrum Allocation Chart*. en-US. URL: https://reach.ieee.org/primary-sources/south-africa-spectrum-allocation-chart/ (visited on 11/01/2021).

# Appendix A

# Frequency Allocation Chart

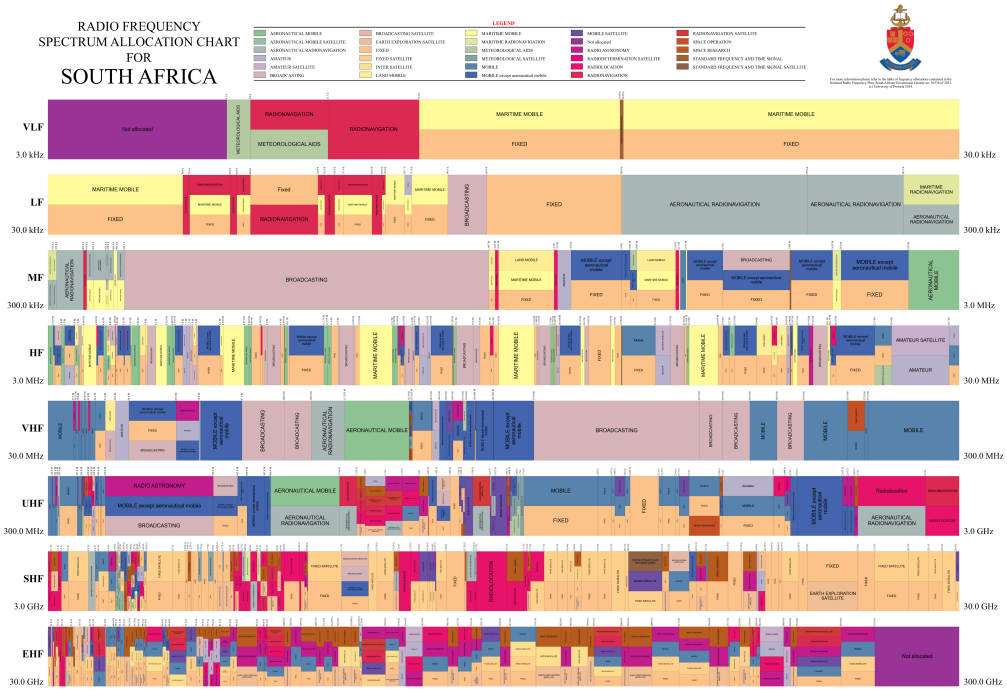



FIGURE A.1: Radio frequency spectrum allocation chart for South Africa [219].

This chart illustrates frequency allocations in the electromagnetic spectrum for various purposes as regulated by the government of South Africa. The Radio Frequency Spectrum Allocation Chart is useful for scientists, astronomers, and engineers seeking to develop policies to protect radio observatories

# Appendix B

# RFI Questionnaire

## 1.0: Requirements collection for RFI database

The questionnaire will gather RFI data from three (3) user categories, namely from engineers, astronomers, and computer scientists. A respondent is required to clearly indicate which user category they belong. This will assist to identify specific requirements for a single group of users.

**1.1: Select your user category**

- ○ User 1: Engineer
- ○ User 2: Astronomer
- ○ User 3: Computer scientist

**1.2: What is the primary source of RFI input data?**

- ○ RTA
- ○ MeerKAT
- ○ KAT7
- ○ Spectrograph
- ○ Other:

**1.3: How would you like to enter the RFI data into the database?**

- ○ Manual entry
- ○ Automatic entry
- ○ Both of them

**1.4: In what file formats is RFI data being collected?**

- ☐ CSV
- ☐ HDF5
- ☐ FITS
- ☐ PDF
- ☐ Measurement sets
- ☐ Other:

**1.5: In what range of file sizes is RFI data collected?**

- ○ Below 1 Gigabytes
- ○ 1 Gigabytes -- 1 Terabytes
- ○ 1 Terabytes -- 1 Petabytes
- ○ Above 1 Petabytes

**1.6: How would you like to classify RFI signals/data collected? (Select all that apply)**

- ☐ by source
- ☐ by shape
- ☐ by transmission (intentional or unintentional)
- ☐ by time
- ☐ Other:

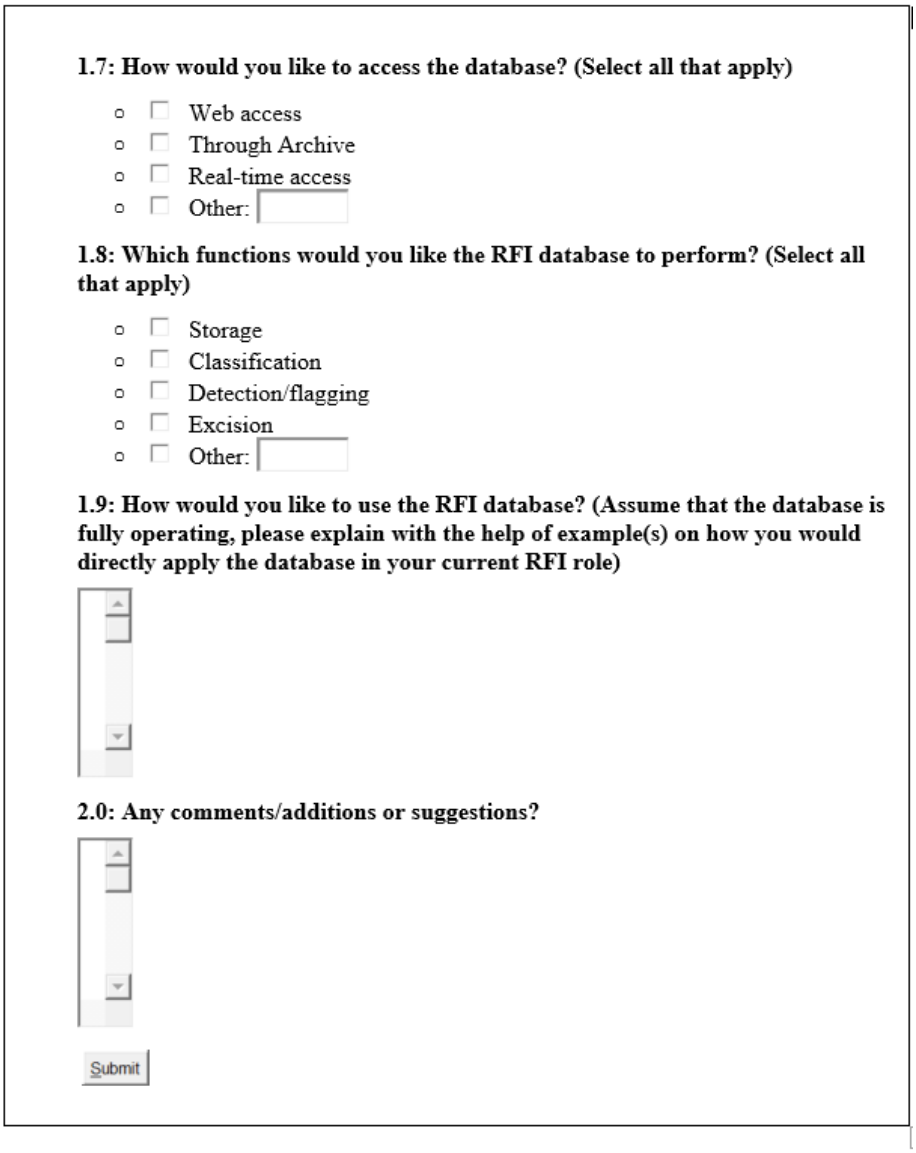



FIGURE B.1: Questionnaire for RFI database requirements gathering.

An electronic questionnaire was created with the help of Google Forms. This was used to gather RFI data across three user categories: engineers, scientists, and astronomers. The data entered was automatically saved to a Google Sheets-based cloud storage for analysis.

# Appendix C

# Sample query syntax and output

## C.1 *Q1: Return all RFI events in a given frequency band (limit to 50 events).*

```
curl -X POST -d "bdarray(filter(project(
        events,eventID,transmitterID,startFreq
        endFreq,elevel,MeerKATBand),
        i<=50 and MeerKATBand <> 'None'));"
        http://192.168.0.117:8080/bigdawg/query/
```

(A)

| eventID | transmitterID | startFreq | endFreq | elevel | MeerKATBand |
|---|---|---|---|---|---|
| 4 | rfi001 | 150 | 200 | 67 | HERA |
| 7 | rfi003 | 30 | 300 | 57 | HERA |
| 9 | rfi004 | 756 | 756 | 35 | UHF\|L-Band |
| 10 | rfi004 | 1607 | 1607 | 35 | L-Band\|S-Band |
| 11 | rfi005 | 2430 | 2430 | 86.1 | S-Band |
| 13 | rfi007 | 1800 | 1800 | 88 | S-Band |
| 14 | rfi008 | 2120 | 2120 | 50 | S-Band |
| 15 | rfi009 | 130 | 130 | 30 | HERA |
| 16 | rfi009 | 199 | 199 | 38 | HERA |
| 18 | rfi010 | 2400 | 2400 | 88 | S-Band |
| 25 | rfi015 | 200 | 200 | 34 | HERA |
| 28 | rfi017 | 92.1 | 105.7 | 44.1 | HERA |
| 29 | rfi017 | 140.55 | 140.55 | 45.1 | HERA |
| 30 | rfi017 | 154 | 154 | 40.2 | HERA |
| 31 | rfi017 | 183.25 | 183.25 | 46 | HERA |
| 34 | rfi018 | 173 | 173 | 50.1 | HERA |
| 35 | rfi018 | 179.2 | 179.55 | 53.3 | HERA |
| 36 | rfi018 | 183 | 185.75 | 55.6 | HERA |
| 38 | rfi018 | 732 | 743.5 | 53.6 | UHF |
| 39 | rfi018 | 937.15 | 937.15 | 57.6 | UHF\|L-Band |
| 40 | rfi019 | 102.1 | 102.1 | 30.9 | HERA |
| 41 | rfi019 | 183.25 | 183.25 | 35 | HERA |
| 43 | rfi020 | 92 | 105.7 | 46.2 | HERA |
| 44 | rfi020 | 935 | 958.6 | 57.6 | L-Band |
| 47 | rfi021 | 935 | 958.6 | 60 | UHF\|L-Band |
| 55 | rfi023 | 175 | 175 | 46.7 | HERA |
| 56 | rfi023 | 199.8 | 199.8 | 37.3 | HERA |
| 59 | rfi023 | 875.5 | 875.5 | 52.3 | UHF |
| 61 | rfi024 | 105 | 105 | 28 | HERA |
| 63 | rfi024 | 980 | 980 | 44 | UHF\|L-Band |
| 68 | rfi025 | 690 | 690 | 52 | UHF |
| 69 | rfi025 | 805 | 805 | 58 | UHF |
| 70 | rfi025 | 850 | 1000 | 50 | UHF\|L-Band |
| 71 | rfi028 | 30 | 230 | 50 | HERA |
| 74 | rfi029 | 180 | 200 | 25 | HERA |
| 75 | rfi030 | 90 | 140 | 83 | HERA |
| 76 | rfi030 | 200 | 200 | 80 | HERA |
| 80 | rfi030 | 400 | 600 | 85 | UHF |
| 81 | rfi030 | 650 | 800 | 84 | UHF |

(B)

FIGURE C.1: A) Query syntax, and B) output of Q1 statement.

The query in Figure C.1a filters out the first 50 RFI events detected in a frequency band other than MeerKAT. It further returns a few columns (eventID, transmitterID, startFreq, endFreq, elevel, and MeerKATBand) specified by a project operator. Figure C.1b displays the query output when executed successfully.

## C.2 *Q2: For a particular RFI event detected, show me related permit and transmitter details.*

```
curl -X POST -d "bdtext(
'op' : 'scan', 'table' : 'transmitter2',
'range':  'start': ['rfi050','permit',''],
'end' : ['rfi050','permit','z'] );"
http://192.168.0.117:8080/bigdawg/query/
```

(A)

```
rfi050  permit:contact1Email    kfones@ska.ac.za
rfi050  permit:contact1Name     kitty jones
rfi050  permit:contact1Organisation     SARAO
rfi050  permit:deployDate       02/06/2015
rfi050  permit:expiryDate       09/09/2020
rfi050  permit:issueDate        02/06/2015
rfi050  permit:notes    unknown
rfi050  permit:permitID RFI1807-0014-050
rfi050  permit:restrictionLimited       {Zone 1|Zone 2}
rfi050  permit:restrictionNo    Zone 0 Within 20m from Antenna
rfi050  permit:restrictionUnlimited     {Zone 3|Zone 4| Zone 5|Zone 6|Zone 7| Zone 8 |Zone 9|Zone 4| Zone 5}
rfi050  permit:rfiNotice        Type A
rfi050  permit:rfiZoneMap       /home/images/rfizone.png
rfi050  permit:type     permanent
rfi050  permit:usage    inhouse
rfi050  permit:useAfterTime     16h00
rfi050  permit:useBeforeTime    14h00
rfi050  permit:useDay/night     day only
rfi050  receiver:adcType        unknown
rfi050  receiver:bandwidth      200
rfi050  receiver:bitstream      [unknown,unknown]
rfi050  receiver:direction      Horizontal,Vertical]
rfi050  receiver:highFreq       [2300,1600]
rfi050  receiver:location       [karoo,karoo]
rfi050  receiver:lowFreq        [180, 900]
rfi050  receiver:nAccs  [10.50,4000]
rfi050  receiver:nChannel       [1024, unknown]
rfi050  receiver:name   [RTA,KAT7]
rfi050  receiver:receiverID     [recv001,recv003]
rfi050  receiver:rfGain [60,100]
rfi050  receiver:spectrumBits   [100,300]
rfi050  transmitter:category    VSAT
rfi050  transmitter:description Vox Telecom VSAT
rfi050  transmitter:manufacturer        Vox Telecom
rfi050  transmitter:model       VSAT
rfi050  transmitter:notes       Main culprit in sub 1 GHz is the switch, which is not part of the VSAT system
rfi050  transmitter:reportComp  MESA
rfi050  transmitter:reportDate  2015-01-25T00:00:00Z
rfi050  transmitter:reportURL   https://drive.google.com/open?id=0B2RwlFv0BXHqd3NIVzA4U3QwcWc
rfi050  transmitter:transmitterID       rfi050
rfi050  transmitter:type        culprit
```

(B)

FIGURE C.2: A) Query syntax, and B) output of Q2 statement.

The query in Figure C.2a fetches all data associated with the key 'rfi050' and filters out in ascending order the column families starting with 'permit' and stopping at column families that end with 'z'. This is why receiver and transmitter data values associated with the key are also returned, as they come before 'z' in alphabetic order. Figure C.2b displays the query output when executed successfully.

## C.3 *Q3: Return all RFI data objects together with their associated attributes.*

```
curl -X POST -d "bdcatalog(select
a.attributeNo,a.attributeName,o.objectID,
o.objectName from rfi.attribute a, rfi.object o
where a.objectID=o.objectID limit 30);"
http://192.168.0.117:8080/bigdawg/query/
```

(A)

| attributeNo | attributeName | objectID | objectName | description |
|---|---|---|---|---|
| 1 | attributeNo | 1 | attribute | attribute number |
| 2 | attributeName | 1 | attribute | name of attribute |
| 3 | dataType | 1 | attribute | attribute data type |
| 4 | description | 1 | attribute | attribute description |
| 5 | engineID | 2 | engine | engine identification number |
| 6 | engineName | 2 | engine | db engine name |
| 7 | host | 2 | engine | hostname of the database |
| 8 | port | 2 | engine | port number at the host machine |
| 9 | connectionProperties | 2 | engine | connection details (engine version) |
| 10 | databaseID | 3 | database | database identification number |
| 11 | databaseName | 3 | database | database name |
| 12 | objectID | 4 | object | data object identification number |
| 13 | objectName | 4 | object | data object name |
| 14 | receiverID | 5 | receiver | receiver identification number |
| 15 | receiverName | 5 | receiver | receiver name |
| 16 | nChannel | 5 | receiver | designated number of channels |
| 17 | bitstream | 5 | receiver | nature of bits streams |
| 18 | nAccs | 5 | receiver | Number of accumulations per spectra |
| 19 | adcType | 5 | receiver | type of analog-to-digital converter (adc) |
| 20 | spectrumBits | 5 | receiver | number of spectrum bits per channel |
| 21 | rfGain | 5 | receiver | receiver antenna gain |
| 22 | bandwidth | 5 | receiver | receiver bandwidth |
| 23 | lowFreq | 5 | receiver | receiver lowest frequency spectrum |
| 24 | highFreq | 5 | receiver | receiver highest frequency spectrum |
| 25 | rxlocation | 5 | receiver | name of receiver location |
| 26 | rxdirection | 5 | receiver | receiver direction |
| 27 | transmitterID | 6 | transmitter | transmitter identification number |
| 28 | txType | 6 | transmitter | type of transmitter |
| 29 | category | 6 | transmitter | short transmitter description |
| 30 | txDescription | 6 | transmitter | detail transmitter description |

(B)

FIGURE C.3: A) Query syntax, and B) output of Q3 statement.

The query in Figure C.2a provides details of all attributes identified by attributeNo and their corresponding objects. The query performs a Cross-join or Cartesian product and returns only those rows whose object Ids match. Figure C.2b displays the query output when executed successfully

## C.4 *Q4: Compute the number of events detected in a given MeerKAT band.*

```
curl -X POST -d "bdarray(
aggregate(filter(project(events,MeerKATBand),
MeerKATBand='L-Band'),count(*)));"
http://192.168.0.117:8080/bigdawg/query/
```

(A)

| MeerKATBand | count(*) |
|---|---|
| L-Band | 262 |

(B)

FIGURE C.4: A) Query syntax, and B) output of Q4 statement.

The query in Figure C.2a is an aggregation query that returns the total number of RFI events detected in the L-Band of MeerKAT. It also includes the MeerKAT field as the column to be returned alongside the computed single value. Figure C.2b displays the query output when executed successfully.

## C.5 *Q5: Return RFI data whose transmitter is unknown.*

```
curl -X POST -d "bdtext(
        { 'op' : 'scan', 'table' :
        '(grep 'unknown' -b rfi001 -e rfi990 >
        table1)'});"
        http://192.168.0.117:8080/bigdawg/query/
```

(A)

```
rfi590 receiver:adcType []    [unknown,unknown]
rfi590 receiver:bandwidth []    [200,300]
rfi590 receiver:bitstream []    [unknown,unknown]
rfi590 receiver:direction []    [Horizontal,Vertical]
rfi590 receiver:highFreq []    [2300,1600]
rfi590 receiver:location []    [karoo, karoo]
rfi590 receiver:lowFreq []    [180,900]
rfi590 receiver:nAccs []    [10.50,4000]
rfi590 receiver:nChannel []    [1024, unknown]
rfi590 receiver:name []    [RTA,KAT7]
rfi590 receiver:receiverID []    [recv001,recv003]
rfi590 receiver:rfGain []    [60,100]
rfi590 receiver:spectrumBits []    [100,300]
rfi590 transmitter:category []    unintentional
rfi590 transmitter:description []    intense spikes
rfi590 transmitter:notes []    no knowledge of the source of rfi or transmitter
rfi590 transmitter:reportComp []    SARAO
rfi590 transmitter:reportDate []    2020-01-25T00:00:00Z
rfi590 transmitter:reportID []    M20200125 UNK 001
rfi590 transmitter:reportURL []    https://drive.google.com/open?id=0B2RwlFv0BXHqd3NIVzA4U3QwcWc
rfi590 transmitter:txID []    rfi590
rfi590 transmitter:type []    unknown
rfi591 receiver:adcType []    200
rfi591 receiver:bandwidth []    150
rfi591 receiver:bitstream []    300
rfi591 receiver:direction []    Vertical
rfi591 receiver:highFreq []    3000
rfi591 receiver:location []    karoo
rfi591 receiver:lowFreq []    790
rfi591 receiver:nAccs []    400
rfi591 receiver:nChannel []    2500
rfi591 receiver:name []    MeerKAT7
rfi591 receiver:receiverID []    recv002
rfi591 receiver:rfGain []    80
rfi591 receiver:spectrumBits []    2000
rfi591 transmitter:MeerKATBand []    [HERA, S-Band, L-Band, None]
rfi591 transmitter:category []    unintentional
rfi591 transmitter:description []    short-lived spikes
rfi591 transmitter:txID []    rfi591
rfi591 transmitter:type []    unknown
```

(B)

FIGURE C.5: A) Query syntax, and B) output of Q5 statement.

The query in Figure C.2a searches for transmitters with the keyword 'unknown' using keys rfi001 to rfi990. It further scans the query result to display RFI data using the selected keys. Figure C.2b displays the query output when executed successfully.

## C.6 *Q6: Retrieve relational data associated with the culprit transmitter (rfi0401) and display it as key-value data.*

```
curl -X POST -d "bdtext(
        { 'op' : 'scan', 'table' : 'bdcast(
        bdrel(select * from testCulprit
        where transmitterID = "rfi0401"),
        res,'', text)'})"
        http://192.168.0.117:8080/bigdawg/query/
```

(A)

```
208139  1:transmitterID  rfi0401
208139  10:reportcomp    MESA
208139  11:location1     Lynxkolk
208139  12:location1lat (-30.438762)
208139  13:location1long         21.120238
208139  14:location2     Izakshoop
208139  15:location2lat (-30.584627)
208139  16:location2long         21.060213
208139  17:location3     Klipkolk
208139  18:location3lat (-30.438867)
208139  19:location3long         21.120959
208139  2:type  culprit
208139  3:category       Electric Fence
208139  4:manufacturer   unknown
208139  5:model unknown
208139  6:testfacility   Reverb Chamber
208139  7:testdate       2013-01-20T00:00:00Z
208139  8:reportdate     2013-03-27T00:00:00Z
208139  9:reporturl      https://drive.google.com/
                         file/d/0B2RwlFv0BXHqZTN1QVc5WDJWbGM
```

(B)

FIGURE C.6: A) Query syntax, and B) output of Q6 statement.

The query in Figure C.2a is a cross-island query that retrieves sample data from relational data and displays it as key-value data. It fetches relational data for a specific transmitter (rfi0401) and casts the query result into a text island for display. Figure C.2b displays the query output when executed successfully.

## C.7 *Q7: Retrieve RFI event array data along with description of each attribute or metadata associated with the array data.*

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

public class CrossIslandJoinClass implements Runnable {
    private final CountDownLatch thread_start;
    private final CountDownLatch thread_end;
    private String threadName;

    CrossIslandJoinClass(CountDownLatch th_start, CountDownLatch
th_end, String name) {
        thread_start = th_start;
        thread_end = th_end;
        threadName = name;
    }

    public void run() {
        long startTimeMillis = System.currentTimeMillis();
        //conversion start time to seconds
        long startTime_Seconds = TimeUnit.MILLISECONDS.toSeconds
(startTimeMillis);
        threadName = Thread.currentThread().getName();
        System.out.println("Running " + threadName);
        //Waiting to start the thread
        try {
            thread_start.await();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        //thread workload assigned to a userQuery string variable.
        String userQuery = "curl -X POST -d b̈darray(project(filter(
events_object,index>=618 and index<=623),+"startFreq,endFreq,txID,
elevel,measDist,eirp,MeerKATBand,pol,eventID));ï" +
"http:/192.168.0.117:8080/bigdawg/query/>" + threadName + ".txt";

        //create a process builder object to run the query in
a new cmd terminal.
        ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/c", userQuery);
        //if set to true, process builders merges all process to standard
in case of any error output.
        pb.redirectErrorStream(true);
        try {
```

```
            //starting new process using Process object
            Process p = pb.start();
            p.waitFor();
        } catch (IOException | InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        //creating a linesList from ArrayList object
        ArrayList<String> linesList = new ArrayList<>();
        //creating a character-stream from a text file
        String line;
        try (BufferedReader breader = new BufferedReader(new FileReader
(threadName + ".txt"))) {
            // Reading each lines of stream and adding them to the list
            while ((line = breader.readLine()) != null) {
                linesList.add(line);
            }
        } catch (IOException e) {
            //print error message
            System.out.println("text reading interrupted.");
        }
        //string array stores one or more text separated by a single space
        String[] arraydata = linesList.get(0).split("//s+");
        //assign different thread workload to query command
        threadName = Thread.currentThread().getName();
        System.out.println("Running " + threadName);
        userQuery = "curl -X POST -d b̈drel(select * from attributes_object;)ï
+ "  http:/192.168.10.117:8080/bigdawg/query/>" + threadName + ".txt";
        pb = new ProcessBuilder("cmd.exe", "/c", userQuery);
        pb.redirectErrorStream(true);
        try {
            //starting new process using Process object
            Process p = pb.start();
            p.waitFor();
        } catch (IOException | InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        //create a hashmap object 'record' to store key and value
of type string
        HashMap<String, String> record = new HashMap<>();
        try (BufferedReader breader = new BufferedReader
(new FileReader(threadName + ".txt"))) {
            while ((line = breader.readLine()) != null) {
                String[] relationaldata = line.split("");
                record.put(relationaldata[2].trim(), relationaldata[4].trim());
            }
        } catch (IOException e) {
            System.out.println("text reading interrupted.");
        }
        System.out.println("attributeName:description");
        for (String arraydatum : arraydata) {
            //check if attribute is common/equal in array and relational data
```

```
            if (record.containsKey(arraydatum)) {
                //concatenate the data values from array and relational data
                System.out.println(arraydatum + ":" + record.get(arraydatum));
            } else
                System.out.println(arraydatum + "" + "A match does not exist");
            }
        }
        //loop to display each record as a new line
        for (String s : linesList) {
            System.out.println(s);
        }
        long endTimeMillis = System.currentTimeMillis();
        //conversion end time to seconds
        long endTime_Seconds = TimeUnit.MILLISECONDS.toSeconds
(endTimeMillis);
        long totalTime = endTime_Seconds - startTime_Seconds;
        //prints the total execution time for each thread
        System.out.println(threadName + "total time:" + totalTime
+ "(in seconds)");
    }
}


import java.util.concurrent.CountDownLatch;
public class TestCrossIslandJoinClass {
    public static void main(String[] args) {
        //change the number of thread here
        int noOfThreads = 1;

        CountDownLatch thread_start = new CountDownLatch(1);
        CountDownLatch thread_end = new CountDownLatch(noOfThreads);
        //loop to start each thread and terminal at noOfThreads
        for (int x = 1; x <= noOfThreads; x++) {
            Thread th = new Thread(new CrossIslandJoinClass(thread_start,
thread_end, String.valueOf(x)));
            th.start();
        }
        //each thread to starting at the same time
        thread_start.countDown();
        try {
            //waiting for each thread to finish
            thread_end.await();
        } catch (InterruptedException e) {
            System.out.println("thread did not finish successfully");
        }
        System.out.println("All threads finished successfully");
    }
}
```

A Java program (CrossIslandJoinClass and TestCrossIslandJoinClass) is used to fetch and match data from two disparate stores (SciDB and PSQL) within the polystore database system.

```
eventID eventSartFreq eventEndFreq transmitterID elevel eirp MeerKATBand
618     265.4   265.4   rfi0621 17.7    -117.0112043 none
619     965.6   965.6   rfi0621 8.9     50.0    UHFband
620     1293.8  1293.8  rfi0621 13.0    -121.7112043 L-band
621     1662.1  1662.1  rfi0621 9.6     -125.1112043 L-band
622     2590.3  2590.3  rfi0621 10.2    -124.5112043 S-band
623     540.1   540.1   rfi0622 8.9     -125.8112043 none

attributeName   :description
eventID         :rfi event identification number
eventStartFreq  :event's start frequency
eventEndFreq    :event's end frequency
transmitterID   :transmitter identification number
elevel          :effective radiated power of an event (in dB)
eirp            :equivalent isotropically radiated power of an event (in dbW)
MeerKATBand     :MeerKAT band an event went off
```

(A) Output of Q7 (procedural query).

The query in Figure C.7a is a procedural query that retrieves data particularly from PSQL and SciDB stores. It consists of four major procedures: executes the query command to retrieve array data, while specifying eventID, eventStartFreq, transmitterID, elevel, eirp, and MeerKATBand attributes; For each attribute that is returned, we check if it exists in the catalog or metadata; If a match is found; we then return the attributeName along with its description.

# Appendix D

# Tables showing plot data

## D.1 *Store response time, latency, download speed and upload speed as the numbers of records increase from 200 to 25600.*

| | 200 | 400 | 800 | 1600 | 3200 | 6400 | 12800 | 25600 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 0.060443 | 0.087410 | 0.121058 | 0.147669 | 0.304818 | 0.518049 | 1.211063 | 1.847499 |
| **SciDB** | 1.757230 | 1.799992 | 1.832895 | 2.194224 | 2.537502 | 3.168941 | 4.327154 | 8.155505 |
| **Accumulo** | 0.502707 | 0.521180 | 0.538998 | 1.322450 | 1.477809 | 3.032555 | 8.266367 | 9.941518 |

(A) Response time (seconds)

| | 200 | 400 | 800 | 1600 | 3200 | 6400 | 12800 | 25600 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 0.004134 | 0.003556 | 0.004274 | 0.003329 | 0.003320 | 0.004180 | 0.004048 | 0.004038 |
| **SciDB** | 0.003321 | 0.003232 | 0.003307 | 0.004845 | 0.003285 | 0.003247 | 0.003118 | 0.003231 |
| **Accumulo** | 0.003537 | 0.003423 | 0.003460 | 0.004902 | 0.004887 | 0.005731 | 0.006150 | 0.005035 |

(B) Latency (seconds)

| | 200 | 400 | 800 | 1600 | 3200 | 6400 | 12800 | 25600 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 192.283000 | 265.004000 | 381.458500 | 632.764500 | 613.875500 | 722.807000 | 619.590000 | 817.883000 |
| **SciDB** | 7.423000 | 14.544500 | 27.840000 | 48.035500 | 83.036500 | 134.122000 | 196.948500 | 211.180500 |
| **Accumulo** | 17.485500 | 33.226500 | 61.527500 | 46.654500 | 84.899500 | 80.372000 | 58.879500 | 116.803500 |

(C) Download speed (KB/seconds)

| | 200 | 400 | 800 | 1600 | 3200 | 6400 | 12800 | 25600 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 0.235000 | 0.233000 | 0.219000 | 0.089000 | 0.079500 | 0.038500 | 0.014000 | 0.013667 |
| **SciDB** | 0.019000 | 0.018500 | 0.018000 | 0.015500 | 0.013000 | 0.010500 | 0.008000 | 0.004000 |
| **Accumulo** | 1.116000 | 0.770500 | 0.556000 | 0.464500 | 0.224000 | 0.131500 | 0.056500 | 0.037000 |

(D) Upload speed (KB/seconds)

## D.2 *Store response time, latency, download speed and upload speed as the numbers of users increase from 1 to 36.*

| | 1 | 3 | 6 | 12 | 18 | 24 | 30 | 36 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 1.105950 | 0.967733 | 2.154176 | 2.750947 | 2.839452 | 2.945160 | 3.011403 | 3.188398 |
| **SciDB** | 4.327154 | 5.427986 | 6.477691 | 7.643643 | 8.008925 | 10.004285 | 10.285112 | 12.368173 |
| **Accumulo** | 1.136875 | 1.300725 | 3.055347 | 3.296703 | 3.990845 | 4.671495 | 5.761101 | 8.400246 |

(A) Response time (seconds)

| | 1 | 3 | 6 | 12 | 18 | 24 | 30 | 36 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 0.030956 | 0.052260 | 0.072062 | 0.044548 | 0.032768 | 0.022317 | 0.023763 | 0.024623 |
| **SciDB** | 0.017252 | 0.041356 | 0.047184 | 0.029047 | 0.024639 | 0.027013 | 0.019002 | 0.026504 |
| **Accumulo** | 0.025937 | 0.047819 | 0.065101 | 0.040479 | 0.029795 | 0.029090 | 0.023088 | 0.023090 |

(B) Latency (seconds)

| | 1 | 3 | 6 | 12 | 18 | 24 | 30 | 36 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 1224.657500 | 756.076300 | 578.237250 | 446.448917 | 431.539700 | 408.008842 | 412.333214 | 392.406006 |
| **SciDB** | 206.047000 | 197.304200 | 157.499556 | 140.252500 | 128.461367 | 118.432125 | 138.936500 | 98.050871 |
| **Accumulo** | 801.097000 | 668.593000 | 355.084800 | 353.752636 | 270.633200 | 231.775760 | 197.172837 | 141.314708 |

(C) Download speed (KB/seconds)

| | 1 | 3 | 6 | 12 | 18 | 24 | 30 | 36 |
|---|---|---|---|---|---|---|---|---|
| **PSQL** | 0.046000 | 0.028000 | 0.021450 | 0.016354 | 0.015850 | 0.014961 | 0.015100 | 0.014375 |
| **SciDB** | 0.006000 | 0.006000 | 0.004667 | 0.004100 | 0.003767 | 0.003425 | 0.004104 | 0.002786 |
| **Accumulo** | 0.110500 | 0.091700 | 0.043100 | 0.042909 | 0.032767 | 0.027720 | 0.023500 | 0.016694 |

(D) Upload speed (KB/seconds)

## D.3 *Store response time, latency, download speed and upload speed of varying workloads.*

| | Workload A | Workload B | Workload C | Workload D |
|---|---|---|---|---|
| **PSQL** | 1.324644 | 2.243125 | 1.826170 | 3.040785 |
| **SciDB** | 7.622594 | 9.223835 | 8.161582 | 11.257868 |
| **Accumulo** | 2.996443 | 6.062866 | 2.947903 | 6.075333 |

(A) Response time (seconds)

| | Workload A | Workload B | Workload C | Workload D |
|---|---|---|---|---|
| **PSQL** | 0.143100 | 0.049117 | 0.018912 | 0.059809 |
| **SciDB** | 0.021620 | 0.064115 | 0.057768 | 0.072757 |
| **Accumulo** | 0.103071 | 0.094312 | 0.007992 | 0.071721 |

(B) Latency (seconds)

| | Workload A | Workload B | Workload C | Workload D |
|---|---|---|---|---|
| **PSQL** | 442.518000 | 492.588500 | 636.258500 | 382.025000 |
| **SciDB** | 93.604500 | 147.206500 | 174.827500 | 120.305500 |
| **Accumulo** | 161.932000 | 152.206500 | 335.753500 | 161.954500 |

(C) Download speed (KB/seconds)

| | Workload A | Workload B | Workload C | Workload D |
|---|---|---|---|---|
| **PSQL** | 0.051000 | 0.029000 | 0.023500 | 0.022000 |
| **SciDB** | 0.004000 | 0.003000 | 0.004000 | 0.002500 |
| **Accumulo** | 0.039000 | 0.019000 | 0.040500 | 0.020000 |

(D) Upload speed (KB/seconds)

# Appendix E

# *The relation schema used to create the alternative database model.*

```
CREATE TABLE IF NOT EXISTS rfi.receiver (
       receiverID varchar(25) PRIMARY KEY,
       receiverName varchar(50) not null,
       nChannel double not null,
       bitstream varchar(50),
       nAccs integer,
       adcType varchar(50),
       spectrumBits double,
       rfGain double,
       bandwidth double not null,
       lowFreq double not null,
       highFreq double not null
);

CREATE TABLE IF NOT EXISTS rfi.rxLocation (
       rxLocationID serial,
       rxLocationName varchar(50) not null,
       rxDirection varchar(50),
       PRIMARY KEY (rxLocationID),
       receiverID varchar(25) REFERENCES rfi.receiver(receiverID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.rfievent (
       eventID varchar(25) PRIMARY KEY,
       eventStartFreq double not null,
       eventEndFreq double not null,
       elevel double not null,
       measDist double not null,
       eirp double not null,
       MeerKATBand varchar(50),
       polarization varchar(50),
       receiverID varchar(25) REFERENCES rfi.receiver(receiverID) ON DELETE CASCADE,
       transmitterID varchar(25) REFERENCES rfi.transmitter(transmitterID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.transmitter (
        transmitterID varchar(25) PRIMARY KEY,
        txType varchar(50) not null,
        category text not null,
        txDescription text not null,
        txNotes text
 );

CREATE TABLE IF NOT EXISTS rfi.band (
        bandCode varchar(25) PRIMARY KEY,
        bandname varchar(50) not null,
        MeerKATBand varchar(50),
        bandstartFreq double not null,
        bandendFreq double not null,
        bandNotes text,
        PRIMARY KEY (bandCode, transmitterID),
        transmitterID varchar(25) REFERENCES rfi.transmitter(transmitterID)
```

```
);

CREATE TABLE IF NOT EXISTS rfi.intentional (
        transmitterID varchar(25) PRIMARY KEY,
        txType varchar(50) not null,
        category text not null,
        txDescription text not null,
        txNotes text,
        status varchar(50) not null,
        startFreq double not null,
        centFreq double,
        endFreq double not null,
        bandwidth double not null,
        latitude varchar(50),
        longitude varchar(50),
        FOREIGN KEY transmitterID REFERENCES rfi.transmitter(transmitterID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.culprit (
        transmitterID varchar(25) PRIMARY KEY,
        txType varchar(50) not null,
        category text not null,
        txDescription text not null,
        txNotes text,
        manufacturer varchar(50),
        model varchar(50),
        serialNumber varchar(50),
        FOREIGN KEY transmitterID REFERENCES rfi.transmitter(transmitterID) ON DELETE CASCADE

);


CREATE TABLE IF NOT EXISTS rfi.testMeasurement (
        testNo serial,
        testFacility varchar(50) not null,
        testDate date not null,
        reportDate date not null,
        reportURL text not null,
        reportID varchar(50) not null,
        reportComp varchar(50),
        reportNotes text not null,
        PRIMARY KEY (testNo, transmitterID),
        receiverID varchar(25) REFERENCES rfi.receiver(receiverID) ON DELETE CASCADE,
        transmitterID varchar(25) REFERENCES rfi.transmitter(transmitterID) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS rfi.occupancy (
        occupancyID serial PRIMARY KEY,
        month int not null,
        nights int not null,
        evenings int not null,
        strong int not null,
        moderate int not null,
        weak int not null,
        PRIMARY KEY (occupancyID, transmitterID),
        testNo serial REFERENCES rfi.testMeasurement(testNo) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS rfi.testLocation (
        testLocationID serial PRIMARY KEY,
        testLocationName varchar(50) not null,
        testLongitude varchar(50),
        testLatitude varchar(50),
        PRIMARY KEY (testLocationID),
        testNo serial REFERENCES rfi.testMeasurement(testNo) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS rfi.unknown (
        transmitterID varchar(25) PRIMARY KEY,
```

```
        txType varchar(50) not null,
        category text not null,
        txDescription text not null,
        txNotes text,
        FOREIGN KEY transmitterID REFERENCES rfi.transmitter(transmitterID) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS rfi.permit (
        permitID varchar(25) PRIMARY KEY,
        rfiNotice varchar(50),
        issueDate date not null,
        expiryDate date not null,
        usage varchar(50) not null,
        permitType varchar(50) not null,
        deployDate date not null,
        permitNotes text,
        transmitterID varchar(25) REFERENCES rfi.transmitter(transmitterID) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS rfi.contactPerson (
        personID varchar(25) not null,
        contactName varchar(100) not null,
        contactOrganisation varchar(100),
        contactEmail varchar(50) not null,
        PRIMARY KEY (permitID, personID),
        permitID varchar(25) REFERENCES rfi.permit(permitID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.restriction (
        restrictionNo varchar(25) PRIMARY KEY,
        restrictionLimited text not null,
        restrictionUnlimited text not null,
        rfiZoneMap text not null,
        useDayOrnight varchar(50) not null,
        useBeforeTime datetime not null,
        useAfterTime datetime not null,
        PRIMARY KEY (restrictionNo),
        permitID varchar(25) REFERENCES rfi.permit(permitID) ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS rfi.engines (
       engineID serial PRIMARY KEY,
       engineName varchar(50) not null,
       host varchar(50) not null,
       port integer not null,
       connectionProperties varchar(100)
);

CREATE TABLE IF NOT EXISTS rfi.databases (
       databaseID serial PRIMARY KEY,
       databaseName varchar(50) not null,
       engineID serial REFERENCES rfi.engines (engineID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.objects (
       objectID serial PRIMARY KEY,
       objectName varchar(50) not null,
       fields text not null,
       databaseID serial REFERENCES rfi.databases(databaseID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS rfi.attributes (
        attributeNo serial PRIMARY KEY,
        attributeName varchar(50) not null,
        datatype varchar(50) not null,
        description text not null,
        objectID serial REFERENCES rfi.objects (objectID) ON DELETE CASCADE

);
```

The relation schema provided in the alternative design is translated from the alternative logical design, which is clearly explained in Chapter 4. The relation schema fully represents the design at a physical or implementation level by defining the types and sizes of each attribute, as well as enforcing both primary and foreign key constraints. These constraints ensure a high degree of integrity at the physical level of the database.

# Appendix F

# *A full list of all attributes created in the RFI database.*

| attribute no. | attribute | data type | description | Object id (oid) |
|---|---|---|---|---|
| 1 | attributeNo | integer | attribute number (also catalog_id) | 1 |
| 2 | attributeName | varchar(50) | name of attribute in a catalog | 1 |
| 3 | dataType | varchar(50) | attribute data type | 1 |
| 4 | description | text | attribute description | 1 |
| 5 | engineID | integer | db engine identification number | 2 |
| 6 | engineName | varchar(100) | db engine name | 2 |
| 7 | host | varchar(100) | hostname of the database | 2 |
| 8 | port | integer | port number at the host machine | 2 |
| 9 | connectionProperties | varchar(100) | connection details (engine version) | 2 |
| 10 | databaseID | integer | database identification number | 3 |
| 11 | databaseName | varchar(50) | database name | 3 |
| 12 | objectID | integer | data object identification number | 4 |
| 13 | objectName | varchar(50) | data object name | 4 |
| 14 | receiverID | integer | receiver identification number | 5 |
| 15 | receiverName | varchar(50) | receiver name | 5 |
| 16 | nChannel | integer | designated number of channels | 5 |
| 17 | bitstream | varchar(50) | nature of bits streams | 5 |
| 18 | nAccs | integer | Number of accumulations per spectra | 5 |
| 19 | adcType | varchar(50) | type of analog-to-digital converter (adc) | 5 |
| 20 | spectrumBits | double | number of spectrum bits per channel | 5 |
| 21 | rfGain | double | receiver antenna gain | 5 |
| 22 | bandwidth | double | receiver bandwidth (applies to transmitters) | 5 |
| 23 | lowFreq | double | receiver lowest frequency spectrum | 5 |
| 24 | highFreq | double | receiver highest frequency spectrum | 5 |
| 25 | location | varchar(50) | name of receiver location | 5 |
| 26 | direction | varchar(50) | receiver direction | 5 |
| 27 | transmitterID | integer | transmitter identification number | 6 |
| 28 | txType | varchar(50) | type of transmitter | 6 |
| 29 | category | text | short transmitter description | 6 |
| 30 | txDescription | text | detail transmitter description | 6 |
| 31 | txNotes | text | comments or notes on a transmitter | 6 |
| 32 | status | varchar(50) | status of operation of a transmitter | 6 |
| 33 | startFreq | double | transmitter start spectrum frequency | 6 |
| 34 | centFreq | double | central spectrum frequency | 6 |
| 35 | endFreq | double | transmitter end frequency | 6 |
| 36 | bandcode | Varchar(50) | band code of a transmitter | 6 |
| 37 | bandname | varchar(50) | band name of a transmitter | 6 |
| 38 | MeerKATBand | varchar(50) | MeerKAT band where transmitter is detected | 6 |
| 39 | bandstartFreq | double | start of the transmitter band | 6 |
| 40 | bandendFreq | double | end of the transmitter band | 6 |
| 41 | bandNotes | Text | comments about the transmitter band | 6 |
| 42 | manufacturer | varchar(50) | transmitter manufacturer | 6 |
| 43 | model | varchar(50) | transmitter model | 6 |
| 44 | serialNumber | varchar(100) | transmitter serial number | 6 |
| 45 | testFacility | varchar(50) | facility where a transmitter is tested before allowed on site | 6 |

| attribute no. | attribute | data type | description | Object id (oid) |
|---|---|---|---|---|
| 46 | testDate | date | date the transmitter is tested (yy/mm/dd) | 6 |
| 47 | reportDate | date | Date the transmitter report is drafted | 6 |
| 48 | reportURL | Text | URL link to the transmitter report | 6 |
| 49 | reportID | varchar(50) | transmitter report identification number | 6 |
| 50 | reportComp | varchar(50) | company that drafted the transmitter report | 6 |
| 51 | reportNotes | Text | comments written on transmitter | 6 |
| 52 | testLatitude | varchar(50) | latitude of transmitter being tested | 6 |
| 53 | testLongititude | varchar(50) | Longitude of transmitter being tested | 6 |
| 54 | testLocation | varchar(50) | Location name of transmitter tested | 6 |
| 55 | permitID | integer | permit identification number | 7 |
| 56 | rfiNotice | varchar(50) | category of transmitter permit | 7 |
| 57 | issueDate | date | permit issue date | 7 |
| 58 | expiryDate | date | permit expiry date | 7 |
| 59 | usage | text | indicates where permits are used or required | 7 |
| 60 | permitType | varchar(50) | permit type e.g., short or long time | 7 |
| 61 | deployDate | date | date the transmitter is permitted on site | 7 |
| 62 | contactName | varchar(50) | contact person in-charge of the transmitter | 7 |
| 63 | contactOrganisation | varchar(50) | contact organization in-charge of the transmitter | 7 |
| 64 | contactEmail | varchar(50) | contact person's email address | 7 |
| 65 | restrictionNo | varchar(50) | 100% restricted zones of a transmitter on site | 7 |
| 66 | restrictionLimited | text | limited restriction zones of a transmitter on site | 7 |
| 67 | restrictionUnlimited | text | unlimited restriction zones of a transmitter on site | 7 |
| 68 | rfiZoneMap | text | URL link to a map showing rfi zones e.g., zone0 to zone09 | 7 |
| 69 | useDay/night | datetime | Time of the day a transmitter is allowed on site | 7 |
| 70 | useBeforeTime | datetime | Before time a transmitter is allowed on site | 7 |
| 71 | useAfterTime | datetime | After time a transmitter is allowed on site | 7 |
| 72 | permitNotes | text | comments on the permitted transmitter | 7 |
| 73 | eventID | integer | rfi event identification number | 8 |
| 74 | eventStartFreq | double | event's start frequency | 8 |
| 75 | eventEndFreq | double | event's end frequency | 8 |
| 76 | measDist | double | measurement distance to the antenna | 8 |
| 77 | elevel | | effective radiated power of an event (in dB) | 8 |
| 78 | eirp | double | equivalent isotropically radiated power of an event (in dBW) | 8 |
| 79 | polarisation | varchar(50) | direction in which an event is observed | 8 |
| 80 | occupancyID | integer | rfi occupancy identification number | 9 |
| 81 | month | varchar(50) | month for which rfi occupancy is computed | 9 |
| 82 | nights | Integer | number of events during the nighttime (after 2 am) | 9 |
| 83 | evening (storms) | Integer | number of events during the evening time (before 2 am) | 9 |
| 84 | strong | integer | number of strong (>10 dB) rfi events | 9 |
| 85 | moderate | integer | number of moderate (>3 dB) rfi events | 9 |
| 86 | weak | integer | number of weak (<3 dB) rfi events | 9 |

# Appendix G

# Embedded schema in JSON Document

```
"transmitterID": "rfi1402-0401",
"receiverID": "recv001",
"category": "terrestrial",
"txDescription": "SABC radio - SAFM",
"bandname": "Broadcasting (FM radio)",
"bandcode": "B Cast",
"MeerKATBand": "Hera",
"bandstartFreq": 87.5,
"bandendFreq": 108",
"bandNotes": "fm broadcasts happens in this band",
"txNotes":
    "txType":
    "type": "Intentional",
    "status": "operational",
    "startFreq": 104,
    "centFreq": 105,
    "endFreq": 107,
    "bandwidth": 0.2,
    "latitude": -22.332,
    "longitude": 29.3322,
    "txLocation": "Cape Town",
```

(A) Intentional transmitter's data schema in JSON's key-value pairs

```
"transmitterID": "rfi1402-0401",
"receiverID": "recv001",
"category": "Electric Fences",
"txDescription": "Electric Fence at Klipkolk and Lynxkolk",
"bandname": "Broadcasting (FM radio)",
"bandcode": "B Cast",
"MeerKATBand": "Hera",
"bandstartFreq": 87.5,
"bandendFreq": 108",
"bandNotes": "fm broadcasts happens in this band",
"txNotes": "N/A"
    "txType":
    "type": "culprit",
    "manufacturer": "N/A",
    "model": "N/A",
    "testFacility": "Reverb Chamber",
    "testDate": "2013-01-20T00:00:00",
    "reportDate": "2013-01-27T00:00:00",
    "reportURL": "https://drive.google.com/file/d/0B2RwlFv0BXHqZTN1QVc5WDJWbGM"
    "reportID": "M2901-0000-001",
    "reportComp": "MESA",
    "reportNotes": "M2901-0000-001",
    "testLatitude": "[-30.438762, -30.584627]",
    "testLongitude": "[21.120238, 21.060213",
    "testLocation": ["Lynxkolk", "Izakshoop"]
```


(A) Culprit transmitter's data schema in JSON's key-value pairs


```
"transmitterID": "rfi1402-0403",
"receiverID": "recv003",
"category": "N/A",
"txDescription": "A/A",
"bandname": "Broadcasting (FM radio)",
"bandcode": "B Cast",
"MeerKATBand": "Hera",
"bandstartFreq": 87.5,
"bandendFreq": 108",
"bandNotes": "fm broadcasts happens in this band",
"txNotes": "no specific information on the transmitter",
"txType": "unknown"
```


(B) Unknown transmitter's data schema in JSON's key-value pairs

## Appendix H

# Accumulo's key-value schema associated to a single key (rfi001)

```
key  : columnFamily : columnQualifier : value
.....................:.................................
insert rfi001  transmitter  : transmitterID rfi1402-0401
insert rfi001  transmitter  : txType culprit
insert rfi001  transmitter  : category "Electric Fence"
insert rfi001  transmitter  : txDescription "Electric Fences at Klipkolk
and Lynxkolk"
insert rfi001  transmitter  : txNotes unknown
insert rfi001  transmitter  : status operational
insert rfi001  transmitter  : startFreq 30
insert rfi001  transmitter  : centFreq 35
insert rfi001  transmitter  : endFreq 40
insert rfi001  transmitter  : bandwidth 20
insert rfi001  transmitter  : latitude [-30.438762, -30.584627]
insert rfi001  transmitter  : longitude [21.120238, 21.060213]
insert rfi001  transmitter  : txLocation [Klipkolk, Lynxkolk]
insert rfi001  transmitter  : manufacturer unknown
insert rfi001  transmitter  : model unknown
insert rfi001  transmitter  : serialNumber unknown
insert rfi001  transmitter  : testFacility "Reverb Chamber"
insert rfi001  transmitter  : testDate 2013-01-20T00:00:00Z
insert rfi001  transmitter  : reportDate 2013-03-27T00:00:00Z
insert rfi001  transmitter  : reportURL
https://drive.google.com/file/d/0B2RwlFv0BXHqZTN1QVc5WDJWbGM
insert rfi001  transmitter  : reportID M2901-0000-001
insert rfi001  transmitter  : reportComp MESA
insert rfi001  transmitter  : reportNotes unknown
insert rfi001  transmitter  : occupancy unknown
insert rfi001  transmitter  : bandCode "B Cast"
insert rfi001  transmitter  : bandname "Broadcasting (FM radio)"
insert rfi001  transmitter  : MeerKATBand Hera
insert rfi001  transmitter  : bandstartFreq 87.5
insert rfi001  transmitter  : bandendFreq 108
insert rfi001  transmitter  : bandNotes "fm broadcasts happen in this band"
insert rfi001  receiver  : receiverID recv001
insert rfi001  receiver  : receiverName RTA
insert rfi001  receiver  : nChannel 1024
insert rfi001  receiver  : bitstream unknown
```

```
insert rfi001  receiver  : nAccs 10.50
insert rfi001  receiver  : adcType unknown
insert rfi001  receiver  : spectrumBits 100
insert rfi001  receiver  : rfGain 60
insert rfi001  receiver  : bandwidth 200
insert rfi001  receiver  : lowFreq 180
insert rfi001  receiver  : highFreq 2300
insert rfi001  receiver  : rxLocation Karoo
insert rfi001  receiver  : rxDirection Horizontal
insert rfi001  permit  : permitID RFI1807-0014-001
insert rfi001  permit  : rfiNotice "Type A"
insert rfi001  permit  : issueDate 01/01/2010
insert rfi001  permit  : expiryDate 01/06/2011
insert rfi001  permit  : usage "site visits"
insert rfi001  permit  : permitType local
insert rfi001  permit  : deployDate 01/02/2011
insert rfi001  permit  : contactName  "Willem Esterhyse"
insert rfi001  permit  : contactOrganisation SARAO
insert rfi001  permit  : contactEmail westerhyse@sarao.ac.za
insert rfi001  permit  : restrictionNo "Zone 0 Within 20m from Antenna"
insert rfi001  permit  : restrictionLimited "Zone 1|Zone 2"
insert rfi001  permit  : restrictionUnlimited "Zone 3|Zone 4| Zone 5|Zone
6|Zone 7| Zone 8 |Zone 9|Zone 4| Zone 5"
insert rfi001  permit  : rfiZoneMap "/home/images/rfizone.png"
insert rfi001  permit  : useDayOrNight "day only"
insert rfi001  permit  : useBeforeTime 08h00
insert rfi001  permit  : useAfterTime 17h00
insert rfi001  permit  : permitNotes "More than 20m away from any MeerKAT
antenna, Only during daytime (see curfew times)"
```

A complete list of key-value schemas associated with a single key (rfi001). The key can be used to easily retrieve data values of any type, including text, logs, images/scans, video, and sensor data.

## Appendix I

# Java code to execute concurrent user requests.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.concurrent.CountDownLatch;

public class ConcurrentRequestsClass {
    private final CountDownLatch thread_start;
    private final CountDownLatch thread_end;
    private String threadName;

    ConcurrentRequestsClass(CountDownLatch th_start, CountDownLatch th_end) {
        thread_start = th_start;
        thread_end = th_end;
    }

    public void run() throws InterruptedException {
        threadName = Thread.currentThread().getName();
        // Waiting for the thread to start
        thread_start.await();
        //starting the thread/request

        for (int testNo = 0; testNo < 15; testNo++) {
            try {
                String userQuery = "curl -X POST -d b̈darray(filter(
events_object,i>0));¨ http://localhost:8080/bigdawg/query/ -w>>"
+ threadName + ".txt";
                // each userQuery is tested 15 times
                Process p = Runtime.getRuntime().exec(userQuery);
                BufferedReader breader = new BufferedReader(new
InputStreamReader(p.getInputStream()));
                String line;
                while ((line = breader.readLine()) != null) {
                    //display the output
                    System.out.println(line);
                }
                // Waiting for the process to finish
                int exitProcess = p.waitFor();
                if (exitProcess == 0) {
```

```
                    System.out.println("process finished successful");
                } else {
                    System.out.println("process interrupted!");
                }
            } catch (IOException | InterruptedException e) {
                System.out.println(e.getMessage());
            }

        }

    }
}

import java.util.concurrent.CountDownLatch;

public class TestConcurrentRequestsClass {
    public static void main(String[] args) {
        //Change the number of concurrent requests here
        int userRequests = 1;
        //latch indicates the one operation needs to complete before
the waiting thread proceeds
        CountDownLatch thread_start = new CountDownLatch(1);
        CountDownLatch thread_end = new CountDownLatch(userRequests);
        //loop to start each user request
        for (int x = 1; x <= userRequests; x++) {
            Thread th = new Thread(new ConcurrentRequestsClass(thread_start,
thread_end).toString());
            th.start();
        }
        // the command lets each thread to start simultaneously
        thread_start.countDown();
        try {
            // waiting for each thread to finish
            thread_end.await();
        } catch (InterruptedException e)
            System.out.println("user request executed successfully");
        }
        System.out.println("All requests executed successfully");
    }
}
```

A Java multi-threaded program (ConcurrentRequestsClass and TestConcurrentRequestsClass) is used to run multiple requests simultaneously. We manually increase the number of user requests after each test has been repeated 15 times. The same code saves each independent test with a separate file name. No specific delay is added, as we start multiple requests at the same time.

# Appendix J

# Basis Interface (RFI database)

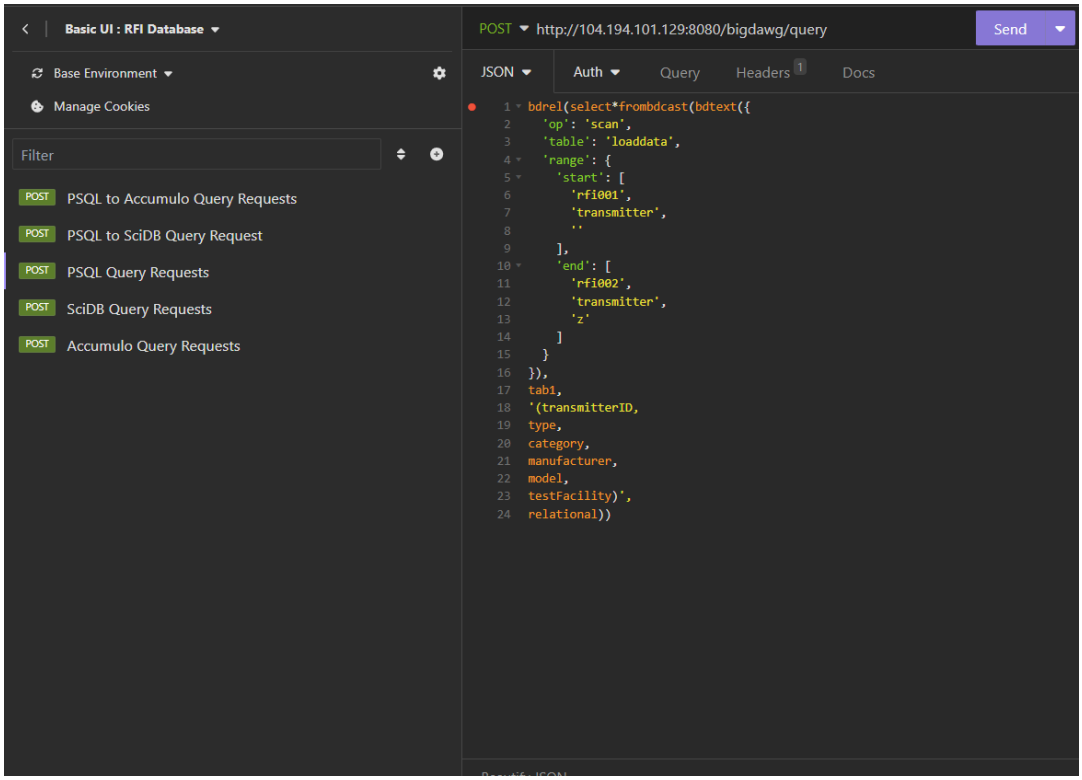



FIGURE J.1: Basic interface: RFI database.

The basic interface uses a web-based tool (Insomnia), also employed in our API evaluation. The tool allows us to create and save customized queries under the POST features. Our cURL query is broken into two parts: 1) the query: http://104.194.101.129:8080/bigdawg/query, and 2) the JSON data. The tool employs the POST request type to send a query along with the data to the database server. We select the customize query and only add or exclude attributes in the JSON structure depending on the requirement of a specific query.