A Scalable Database Model of RFI Data for the MeerKAT Radio Telescope

Gerald Nathan Balekaki
Department of Computer Science
University of Cape Town
Cape Town, South Africa
nbalekaki@cs.uct.ac.za

Michelle Kuttel
Department of Computer Science
University of Cape Town
Cape Town, South Africa
mkuttel@cs.uct.ac.za

Anja Schroeder
South African Astronomical Observatory
Cape Town, South Africa
anja@saao.ac.za

Sarah Blyth
Department of Astronomy
University of Cape Town
Cape Town, South Africa
sblyth@ast.uct.ac.za

Sonia Berman
Department of Computer Science
University of Cape Town
Cape Town, South Africa
sonia@cs.uct.ac.za

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. As RFI corrupts observational data and may even damage radio telescope equipment, astronomers seek to store data on RFI, with the aim of mitigating or preventing future interference events. This is a concern for the MeerKAT telescope, precursor to the Square Kilometer Array and one of the largest and most sensitive radio telescope in the world to date. Currently, RFI data at MeerKAT is collected in many different file formats that do not fit into traditional database models created to store data in a fixed schema. Therefore, we have designed a scalable database model for RFI storage, that supports many databases and many data models. The database is deployed in a Dockerized environment. Preliminary testing of our design shows linearly scaling of data ingestion as data sizes increases, as well as fast query processing.

CCS CONCEPTS

• Information systems → Semi-structured data; **Database management system engines**; **Database design and models**; **• Applied computing → Astronomy**.

KEYWORDS

Radio Frequency Interference, emerging database models, NoSQL, polystores

ACM Reference Format:
Gerald Nathan Balekaki, Michelle Kuttel, Anja Schroeder, Sarah Blyth, and Sonia Berman. 2019. A Scalable Database Model of RFI Data for the MeerKAT Radio Telescope. In *Conference of the South African Institute of Computer Scientists and Information Technologists 2019 (SAICSIT '19), September 17–18, 2019, Skukuza, South Africa.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3351108.3351127

1 INTRODUCTION

The Square Kilometre Array (SKA) project aims to construct the largest and most sensitive telescope in the world, with the first phase (10% of the full SKA) expected to produce early science observations in 2020. The core of the array will be located at the site of South Africa's SKA precursor MeerKAT [1], which has only about 1% of the final SKA's sensitivity but is already one of the most sensitive radio telescopes in the world. The full SKA is expected to produce data volumes equating to the current global total internet traffic [2]. 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 contamination of the radio signal is unfortunately growing due to technological advancements.

The biggest sources of RFI are human generated and often stronger (many billions of times) than the weak celestial signals of interest, drowning them out and corrupting observational data [3]. Some of the potential sources of RFI likely to interfere with radio observations in a typical modern radio observatory are: 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 (DME) (Figure 1), although the majority of RFI sources are unidentified. One source of RFI is from legally allocated communication services licensed by regulatory authorities, such as ICASA in South Africa [4].

Radio astronomers seek to collect, store, and quickly analyse RFI to assist in identifying or removing RFI sources before or during observations, thus keeping the telescope site clean from interference. Currently, there is no complete remedy for RFI, but several approaches have been introduced to reduce the damage on the signal [6, 7]. These approaches range from regulatory to technical methods. The first approach in RFI mitigation is prevention: this is explicitly provided for in the Astronomy Geographical Act (AGA) of South Africa, which protects areas suited to radio astronomy [4]. The second approach involves monitoring and detection of RFI signals. Monitoring of RFI at MeerKAT is implemented using two devices: 1) a fixed monitoring system that provides continuous

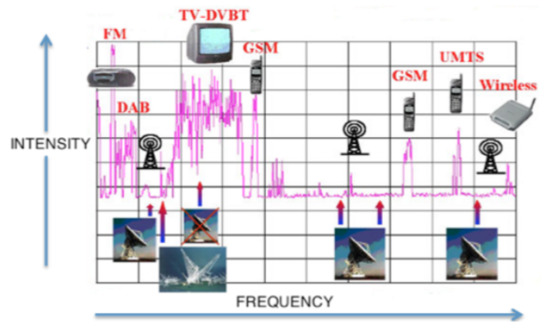
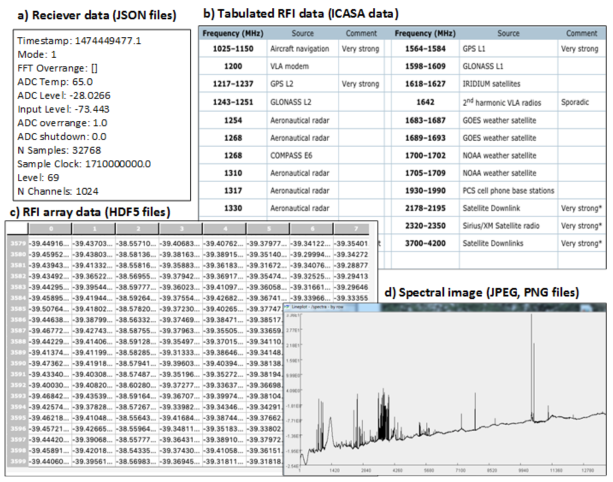


Figure 1: Potential RFI near the typical radio spectrum [5].

monitoring, and 2) a mobile monitoring system that resides on a vehicle that can be deployed anywhere within the telescope array and beyond [8]. Currently, RFI is stored as unstructured data: it is detected, measured and collected at the MeerKAT site in a range of different file formats, including measurement sets, arrays, tables, spectral images and JSON files (Figure 2). 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. However, the variety of formats also creates a storage and access concern for the RFI data. It is clear that working with multiple heterogeneous files limits data analysis and is also very cumbersome. In addition, RFI data collected in isolation (from MeerKAT as well as other international radio telescopes) hinders data coordination [9]. Moreover, this problem will be exacerbated with the huge volumes of data generated at high speeds during large survey observations at the SKA in the future.



a) Reciever data (JSON files)

```
Timestamp: 1474449477.1
Mode: 1
FFT Overrange: []
ADC Temp: 65.0
ADC Level: -28.0266
Input Level: -73.443
ADC overrange: 1.0
ADC shutdown: 0.0
N Samples: 32768
Sample Clock: 1710000000.0
Level: 69
N Channels: 1024
```

b) Tabulated RFI data (ICASA data)

Frequency (MHz)	Source	Comment	Frequency (MHz)	Source	Comment
1025–1150	Aircraft navigation	Very strong	1564–1584	GPS L1	Very strong
1200	VLA modem		1598–1609	GLONASS L1	
1217–1237	GPS L2	Very strong	1618–1627	IRIDIUM satellites	
1243–1251	GLONASS L2		1642	2nd harmonic VLA radios	Sporadic
1254	Aeronautical radar		1683–1687	GOES weather satellite	
1268	Aeronautical radar		1689–1693	GOES weather satellite	
1268	COMPASS E6		1700–1702	NOAA weather satellite	
1310	Aeronautical radar		1705–1709	NOAA weather satellite	
1317	Aeronautical radar		1930–1990	PCS cell phone base stations	
1330	Aeronautical radar		2178–2195	Satellite Downlink	Very strong*
			2320–2350	Sirius/XM Satellite radio	Very strong*
			3700–4200	Satellite Downlinks	Very strong*

c) RFI array data (HDF5 files)

	0	1	2	3	4	5	6	7
3579	-39.44916...	-39.43703...	-38.55710...	-39.40683...	-39.40762...	-39.37977...	-39.34122...	-39.35401
3580	-39.45952...	-39.43803...	-38.58136...	-39.38163...	-39.38915...	-39.35140...	-39.29994...	-39.34272
3581	-39.43943...	-39.41332...	-38.55816...	-39.35883...	-39.36183...	-39.31672...	-39.34076...	-39.28877
3582	-39.43492...	-39.36522...	-38.56955...	-39.37942...	-39.36917...	-39.35474...	-39.32525...	-39.29413
3583	-39.44295...	-39.39544...	-38.59777...	-39.36023...	-39.41097...	-39.36058...	-39.31661...	-39.29646
3584	-39.45895...	-39.41944...	-38.59264...	-39.37554...	-39.42682...	-39.36741...	-39.33966...	-39.33355
3585	-39.50764...	-39.41802...	-38.57820...	-39.37230...	-39.40265...	-39.37747		
3586	-39.44638...	-39.38799...	-38.56332...	-39.37469...	-39.38471...	-39.38517		
3587	-39.46772...	-39.42743...	-38.58755...	-39.37963...	-39.35505...	-39.33659		
3588	-39.44229...	-39.41406...	-38.59128...	-39.35497...	-39.37015...	-39.34110		
3589	-39.41374...	-39.41199...	-38.58285...	-39.31333...	-39.38646...	-39.34148		
3590	-39.47362...	-39.41918...	-38.57941...	-39.39603...	-39.40394...	-39.38138		
3591	-39.43340...	-39.40308...	-38.57487...	-39.35196...	-39.35272...	-39.38194		
3592	-39.40030...	-39.40820...	-38.60280...	-39.37277...	-39.33637...	-39.36698		
3593	-39.46842...	-39.43539...	-38.59164...	-39.36707...	-39.39974...	-39.38104		
3594	-39.42574...	-39.37828...	-38.57267...	-39.33982...	-39.34346...	-39.34291		
3595	-39.46218...	-39.41048...	-38.55643...	-39.41684...	-39.38744...	-39.37662		
3596	-39.45721...	-39.42665...	-38.55964...	-39.34811...	-39.35183...	-39.33802		
3597	-39.44420...	-39.39068...	-38.55777...	-39.36431...	-39.38910...	-39.37972		
3598	-39.45891...	-39.42018...	-38.54335...	-39.37430...	-39.41058...	-39.36151		
3599	-39.44060...	-39.39561...	-38.56983...	-39.36945...	-39.31811...	-39.31818		

d) Spectral image (JPEG, PNG files)

Figure 2: RFI data collected in different formats

In recent decades, radio astronomers have seen the introduction of structured file systems such as HDF5 and NetCDF, as a solution for storing huge and diverse datasets [10]. A key feature of such file systems is that they consist of multi-dimensional arrays that act as granules which encapsulate a number of observations over space and time as a single data unit. This kind of composition does not fit into traditional 2D model of rows and columns [11]. In addition, they lack the ability to provide full functionality of a DBMS in terms of indexing, cross-querying and structured languages. Such tools therefore cannot support data intensive scientific applications like real-time data support, deep analytics, and data visualisation [11, 12]. The limitations of such tools have stimulated the development of innovative database models.

Traditional database models, such as relational database (RDBMS) developed by Codd [13], have been around for close to five decades. Although these have been the bedrock of most data management solutions, they strictly rely on organising data in tables of fixed rows and columns. A key challenge faced by the relational models is the inability to scale and integrate data in multiple formats, without compromising the essential READ and WRITE data operations. This is a problem, because a large component of scientific data in general comprises multi-dimensional structures that are inappropriate for relational models. In particular, the use of a RDBMS for storing RFI would imply that only well-structured data that fits well in tables would be able to be stored, while semi-structured or unstructured data would have to be discarded, in spite of all the valuable information they might contain.

New SQL models such as SciDB have been developed to support scientific applications [14]. They particularly strive to provide horizontal scalability without abandoning SQL and consistency [15]. Horizontal scalability refers to the ability to distribute both the data and the operations over several simple servers, with no sharing of RAM or disk space (also known as *shared nothing*), among the servers.

In 2006, the Google Big Table project led to the birth of non-relational SQL models (i.e NoSQL), also regarded as *not only* SQL, to store and manage large volumes of unstructured data such as user session data from chats; messaging; log data, time series data; video and images [16, 17]. Non-relational SQL models aim for improved scalability and performance over both traditional and new SQL models. A key difference between NoSQL and SQLs models is that NoSQLs isolate data storage and management. This provides improved flexibility over SQLs since they do not rely on normalised data that is fixed in rows and columns [18].

Table 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

Currently emerging database models are based on the concept of using a range of models, databases, and languages to provide for a variety of storage needs (often termed *polyglot persistence*) [19, 20]) as different kinds of data fit well with different data stores. Polystore database systems are relatively new approach that shows promise in addressing the volume challenges and unstructured nature of scientific data [21] in comparison to previous approaches (Table 1). One significant development of a polystore is the capability to match and query data across multiple and distinct storage engines (an engine is a core component of our database management system). The fundamental element for a polystore is a central component (middleware) that supports single modality, with the ability to submit queries that may be executed in different data engines, while supporting the competing notions of location transparency and semantic completeness [22, 23]. Researchers at the Intel Science and Technology Center (ISTC) at MIT have developed a prototype database on the polystore database model [22], but much work is still needed to fully develop the concept into a standard for handling scientific datasets [24].

In this work, we aim to address the RFI data storage challenge at MeerKAT, investigating the use of a polystore database model for storing RFI data. We focus on a developing a scalable model that supports a large and diverse range of data formats and enables rapid search. Such a data store is expected to support subsequent efforts in RFI mitigation at the MeerKAT site, such as source classification and statistical analysis.

2 RELATED WORK

Scientists typically prefer storing data in traditional flat files, such as the CSV format. Flat files have no structure for indexing, which limits cross-file connections. However, there are advantages to this approach: many studies show that it works with small numbers of small files and has a modest storage requirement [11, 12]. It is obviously easier to access or manipulate smaller files with simple file commands (e.g. *ls*, *find*, *vi*, *cat*, or *grep* in Linux/Unix) than rather run a batch of *SQL SELECT* queries. However, as the number or size of files grows, such file manipulation becomes more cumbersome.

Recently more structured files – such as HDF5, HDFS, and NetCDF – have been introduced as a solution for storing large and diverse structures [25, 26]. Although structured files support storage of multi-dimensional data, their underlying hierarchical architecture is still file-based, implying that it is still somewhat cumbersome to combine data from several observations [26].

Currently multi-dimensional array-based databases, such as SciDB and Rasdaman, have been developed with a goal to do for the science community what RDBMS accomplished in the business world [25]. SciDB is a highly scalable database system that supports multi-dimensional array storage and complex analysis. Multi-dimensional array-based databases show enormous potential for the storage and analysis of scientific data, however the challenges lie in the loading of the data and its practical application. For Image analysis, SciDB has been demonstrated to outperform not only traditional DBMS, but also other Big Data tools. A case study is EarthDB, which encapsulates the the Moderate Resolution Imaging Spectroradiometer (MODIS) dataset from NASA [27]. This dataset was generated by orbiting satellites (Terra MODIS and Aqua MODIS) that scan the entire surface of the Earth every 1 to 2 days aross 36 spectral bands, with an aim of understanding of global dynamics and processes occurring on the land, in the oceans, and in the lower atmosphere. EarthDB is built upon the SciDB database platform in order to provide the data model, query language, and user defined features. The EarthDB system was demonstrated to meet key expectations of Earth scientists, including support for massive data loading, faster filtering, unified schema representation, and rapid ad hoc analysis.

Case studies on the polystore model indicate that it is a suitable model for complex scientific datasets. The MIMIC II medical dataset contains unidentified health data collected from thousands of critical care patients in Intensive Care Units, including the 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 (e.g. measurements from bedside devices such as heart rate, pulse) [28]. The metagenomics data set contains data on bacteria found in samples of sea water, to study the relationship between the diversity of marine cyanobacteria such as *Prochlorococcus* and environmental variables [21, 29]. This diverse dataset comprises sample and sensor metadata, genome sequences of about 20 million sequences per sample, cruise reports and streaming sea flow data. For both of these cases, the polystore database model was demonstrated to be effective, with significant speedup over traditional approaches based on a single storage engine [22].

3 DATABASE DESIGN

We began our design process for the RFI database by gathering user requirements. These were then used to develop a conceptual database model, which was followed by a logical model. Finally, we mapped the logical model onto a physical model using a polystore architecture.

3.1 Requirements Gathering

As a preliminary step, we gathered users' requirements from a group of radio astronomers, engineers and computer scientists with the aid of an online questionnaire. We collated responses from 23 participants on large Google spreadsheet, categorising similar responses under the same requirement. These requirements were then mapped onto the database components necessary to support them, as shown in Table 2.

Table 2: A summary of the RFI database user requirements and corresponding required database components.

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

Meeting all the user requirements thus requires a standard database API, integrated data stores, rapid RFI classification algorithms and computing timely statistics on the RFI data. The first two design components have to be achieved first, in order to support the other two. In other words, for rapid classification and timely statistical analysis to be successful, RFI data needs to be stored effectively in a database. Therefore, we decided to focus this work on the first two design components: setting up an integrated RFI database and building a basic interface using a standard RESTful API.

3.2 Conceptual design

Our conceptual model shows a global view of the entire RFI database. It is used to give a relatively low level understanding of the RFI data environment. Our design philosophy is to leave data in its native format, as we model the data to fit the data store. This is useful because it limits unnecessary transformation of data into different formats, hence preventing data deterioration. There is no store for all data structures *"one size does not fit all"*, therefore different data structures require distinct store types [21]. The reason for representing the conceptual model as an embedded structure which appear as single entity is to provide a simple view of data in multiple types and, furthermore, to speed up search results as the number of indexes reduce as both the objects and sub-object use the same index, unlike in conventional structure where each object requires a separate index. This implies that both designers and end-users have fewer objects to deal with, thus reducing the complexity of the data environment.

Our conceptual design follows a top-down design strategy [17], identifying the dataset first, and then defining the required data objects. In this case, our RFI dataset is organised in four main objects with sub-objects (Figure 3). The main objects are RFI receiver, FRI transmitter, RFI event and RFI permit. An RFI permit is a document issued locally by SKA engineers after thorough measurements have been taken on a given device emitting RFI; the permit states the strict conditions under which that device can be used on the telescope site.

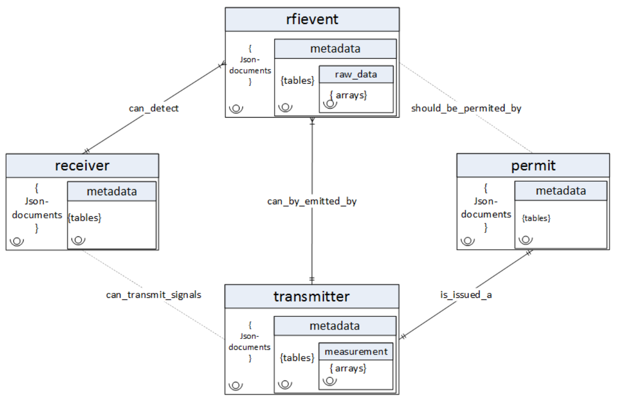


Figure 3: Our conceptual model for an RFI database.

Embedded in each object are sub-objects to store data of similar structure or type, but under the one entity or container. These sub-objects represent data of different modality within an object i.e. relational tables, sparse matrices or array and key-value. All well-structured RFI data, such as the RFI metadata or ICASA transmitter data, is represented in relational tables. Text (such as scientists RFI notes) and images (such as RFI spectral images) are represented as key-value stores. Time series, spectral domain or multi-dimensional data (such as RFI spectra data) can be represented in a sparse matrix model or an array data store.

3.3 Logical design

We represent the logical model with a Unified Modelling Language (UML) class diagram. UML modelling is widely applied as a standard which database designers use to represent data graphically. Further, the UML data modelling method applies well to an object-oriented modelling problem [30], which is the case for many complex datasets, including RFI data. The UML representation of attributes along with the relationships and participation constraints, is helpful when designing the physical schema and supports basic database functionalities like create, read, update and delete (CRUD) operations. At this stage, we translate the conceptual design into the logical model and also show class hierarchy among data objects (Figure 4).

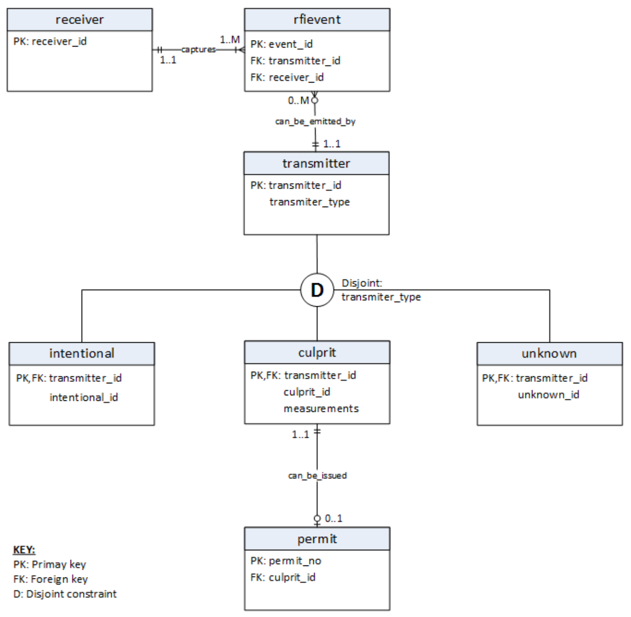


Figure 4: The UML class diagram representing the logical model of our RFI database.

We show the four main objects, but also show three RFI transmitter categories: *intentional*, *culprit* and *unknown*. These sub-objects inherit the properties of transmitter objects via transmitter_id, which acts as both primary and foreign key. Intentional transmitters are licensed nationally and can legally transmit (except in the radio quiet zone), culprit transmitters have been detected at the site and their measurements are taken to establish their harmfulness, these are allowed to transmit under strict permissions, while unknown transmitter is one whose source of transmission is unknown, but we however store related information that might guide us to an accurate identification. Note that a transmitter can only be of one of these three types and only culprits that have been measured and thoroughly studied can be issued permits. The receiver object contains receiver metadata, and uniquely identifies each record by receiver_id. The RFI event object stores event details, such as flags, timestamps and the frequencies at which these events have been detected.

Our logical model also represents the design constraints set by astronomers and engineers regarding RFI data environment at MeerKAT. For example, a RECEIVER is more likely to capture one or more RFI EVENTs, whereas an EVENT may be captured by one or more RECEIVERs; an RFI EVENT belongs to a TRANSMITTER of either type intentional, culprit or unknown, but each TRANSMITTER may emit several RFI EVENTs. A PERMIT can be issued to a specific TRANSMITTER of type CULPRIT, although each CULPRIT may or may not have a PERMIT. We formulate a sample of representative queries based on user requirements analysis.

3.3.1 Q1: Return all RFI events in a given band or time period. The essence of the query is to find the existing RFI events in a given frequency band or at a particular time period at the site. We specify a time period or frequency band and return all possible events both from known sources and unknown sources.

3.3.2 Q2: For a particular RFI event detected, show me related permit and transmitter details. The output of the query seeks to provide astronomers with the permit detail for a particular RFI event, and its likely source.

3.3.3 Q3: Given a receiver, return all captured RFI events in a given time frame or frequency band. The essence of the query seek to enable users to study the behaviour of RFI events that are captured by two or more receivers.

3.3.4 Q4: Return all RFI events whose transmission is unknown. This process assists astronomers to quickly build related information on any unknown events.

3.4 Physical model: polystore architecture

A polystore architecture spans many data stores, and many data models. We adopt the polystore prototype developed at MIT known as BIGDAWG polystore [22]. The bigdawg store comes has three categories for data stores: PSQL, Accumulo, and SciDB. RFI data is stored in each of the data stores, while being exposed to an integrated API. The polystore communicates to each of these stores the help of *shims*, which translate transactions to and from a common middleware language, whereas *casts* are used to move datasets or intermediate results from one system to another as a result of cross-platform querying. In other words, polystore queries select the system that will be responsible for executing different clauses in a query.

Figure 5 illustrates the polystore architecture that we used for RFI storage. We categorise the RFI dataset depending on its native structure (i.e. structured, semi or unstructured). All structured data, like RFI metadata, is stored in relational tables with Postgres (PSQL). Semi or unstructured RFI data (such as text, images and reports) are stored in key-value stores with accumulo. Spectral and time series data are stored in the SciDB array database. Engineers load the data during RFI monitoring and detection, while scientists can perform quick searches and detailed analyses.

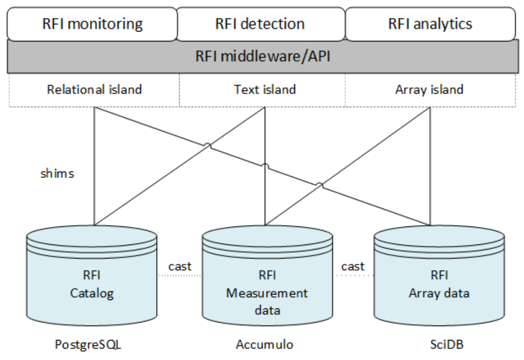


Figure 5: A polystore database prototype for storing RFI data.

4 IMPLEMENTATION

We use Docker containers to deploy the database model. Docker is a container-based virtualisation technology where isolated applications are created, deployed and executed [31]. This technology enables creation of a system that houses several Docker containers working under the same cluster and able to interact with each other or even between applications. Docker containers are preferable to a Virtual Machine (VM) because several containers can run on the same host and share the OS kernel with other containers, without affecting the host machine. A VM environment requires more time and space, since it has to fully boot the individual OS kernel per VM. Docker has other advantages too: it is portable (an application and all its dependencies can be bundled into a single container that is independent from the host OS kernel version, platform distribution or deployment model), lightweight (Docker requires minimal resources, as its images are very small) and fast. This facilitates rapid delivery and reduces the time to deploy an application. Docker is considered to be the most efficient environment for running multiple applications which suffer from integration and interoperability complexities [32].

We use Docker (version 18.09.3, build 774a1f4) to house three databases applications as single network cluster, where multiple containers are distinct database systems, on a single Ubuntu machine of 8GB RAM and 1TB disk capacity. We set up the polystore cluster using bigdawg as provided on the github repository. The cluster comes with a number of containers, each with a distinct image of the database application. Docker helps us to emulate an environment where RFI data is processed using several applications and stored in many locations or machines. This setup is the most suitable option for deploying the RFI database model.

4.1 Storage setup

Each data store is setup using Docker containers with container_id, image, status and ports on our prototype bigdawg cluster, as follows.

```
CONTAINER ID        IMAGE               STATUS        PORTS
-----------------------------------------------------------------------
00c7b470b91b   bigdawg/accumulo   Up 2 days     0.0.0.0:42424->42424/tcp
bda4cdd25c55   bigdawg/accumulo   Up 2 days     0.0.0.0:9999->9999/tcp,
0.0.0.0:50095->50095/tcp
f603766fc514   bigdawg/accumulo   Up 2 days     0.0.0.0:9997->9997/tcp
7f6028a02169   bigdawg/accumulo   Up 2 days     0.0.0.0:2181->2181/tcp
0acad648c758   bigdawg/accumulo   Up 2 days
a0205e6d792f   bigdawg/scidb      Up 2 days     0.0.0.0:1239->1239/tcp
f42f3eccc818   bigdawg/postgres   Up 2 days     0.0.0.0:5402->5402/tcp
120e6d7700ea   bigdawg/postgres   Up 2 days     0.0.0.0:5401->5401/tcp
b6ce75af201e   bigdawg/postgres   Up 2 days     0.0.0.0:5400->5400/tcp,
0.0.0.0:8080->8080/tcp
```

Each container must be up and running to enable access of data within, or between, containers. The bigdawg cluster comes with list of 9 containers of three database categories i.e postgres, scidb or accumulo. Being a cluster of containers, we can have as many containers as possible and we can tailor each container as a data store to match our data requirement. We keep record of the data store or container_id as we create as many databases and data objects as possible. This is helpful especially when queries are fetching data, as a transaction is mapped by store and engine_id. Each is container is identified by the data store and engine number. Table **??** shows a list of data stores with their corresponding engines. We have up to six data stores and four engines, each identified by dbid and engine_id, respectively. Each data store can be accessed locally or remotely through the RESTful API, using CURL commands.

dbid	engine_id	name	userid	password
0	0	rfi_catalog	postgres	test
1	0	rfi_schemas	postgres	test
2	1	rfi_data1	postgres	test
3	2	rfi_data2	postgres	test
4	3	scidb_local	scidb	scidb123
5	4	accumulo	bigdawg	bigdawg

We also set up the catalog to store not only RFI data, but also cluster information or a system catalog. For example, inside a system catalog we derive information on engines, databases, data models whereas data about RFI data objects can also be found in the same catalog. In practice, most of the RFI data is stored in the accumulo and scidb data store. For example, in accumulo data store, we maintain 2500 rows of data, each row has a key, columnfamily, columnqualifier, timestamp and value. For scidb, we maintain four 2-dimensional array tables i.e. 1st array matrix 158,351 by 8, 2nd 316,703 by 8, 3rd 633,407 by 8 and 4th 1,266,814 by 8.

Table 3: showing database objects each location

oid	object_name	logical_db	physical_db
1	rfi1.metadata	0	1
2	rfi1.permit	4	5
3	rfi1.transmitter	4	5
4	rfi1.receiver	4	5
5	rfi1.rfievent	4	5
6	rfi1.measurement	3	4

The physical_id specifies the database number where the object resides, while the logical_id signifies the number of the engine of the corresponding data store. For example, transmitter data can be found in the accumulo data stores running on the acummulo engine. RFI metadata is found in the catalog database running on postgres. The importance of providing location information about each data object in the database is to ensure a self-describing database.

4.2 Standard API

The Docker API is set up on the docker containers to be accessible remotely as well as locally. We test the API's ability to pick data from a native data model (native-island) and across different models (i.e. cross-island). We use the standard RESTful CURL command to POST our query criteria in form of data through a URL to and from a docker cluster, while specifying a client's local or remote IP address and port number. For example:

4.2.1 Native island query: Return all transmitters and their related permits details like issue and expiry date. The essence of the query is to assist astronomers to retrieve information on transmitters that are not permitted to transmit, and, if allowed, look up the permit restrictions.

```
curl -trace-time -X POST -d "bdtext(
        'op' : 'scan', 'table' : 'testdb1', 'range' :
        'start' : ['t001','',''], 'end' : ['t004','',''] );"
        http://localhost:8080/bigdawg/query/
```

The query will post data to respective data objects with the help information stored in the islands. In this case, the data returned is categorised by column family of transmitter type i.e. culprit, intentional or unknown. Some of the information returned includes transmitter's name, type, band, permit's expiry date as well as transmitter gps location.

```
t001 culprit:band hera
t001 culprit:endfreq 120
t001 culprit:name Electric Fences at Klipkolk and Lynxkolk
t001 culprit:number cul01
t001 culprit:permitid pmt01
t001 culprit:reportdate 27/03/2013
t001 culprit:reporturl https://drive.google.com/open?id=0B2RwlFv0BXHqZTN1QVc5WDJWbGM
t001 culprit:startfreq 150
t001 gps:Klipkolk -30.438867, 21.120959
t001 gps:Lynxkolk 30.438762, 21.120238
t001 permit:certificate A
t001 permit:startdate 01/01/2013
t001 permit:startexpiry 01/01/2014
t002 intentional:band Broadcasting (FM radio) band
t002 intentional:endfreq 108
t002 intentional:name sabc radio - KFM
t002 intentional:number int01
t002 intentional:startfreq 87.5
t003 unknown:band broadcasting
t003 unknown:endfreq 109
t003 unknown:name unknown
t003 unknown:notes this transmitter is unknown since 2009
t003 unknown:number unk01
t003 unknown:startfreq 106
```

4.2.2 Cross-island query: Display each transmitter with measurement details, where transmitter type is culprit. The purpose of this query is to assist in assessing the measurement carried out on each culprit transmitter before it is issued a permit and allowed on the site.

```
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/
```

The query will first filter from culprit measurement array data, then cast the resultant into a relational table showing culprit number, test number, start frequency, end frequency and frequency band. In this case, the query utilises two different islands i.e the array and relational in which the final result is displayed.

```
 culpritNo   |   testNo   |     startFreq   | endFreq    | band
------+-----------+-----------------+----------+----------
     c001    |          1 | 30              | 40         | none
     c001    |          2 | 80              | 70         | none
     c001    |          3 | 90              | 80         | none
     c001    |          4 | 180             | 67         | Hera
     c002    |          1 | 30              | 300        | none
     c002    |          2 | 756             | 756        | none
```

5 RESULTS

We test the database on prototype for query speed and ingest speed. Ingest speed refers to the speed at which data of different sizes is loaded into a table or data object. Table 4 illustrates the number of inserts an object can take per second. We conducted four tests to measure how the model scales as the data size grows. In the first test, test1, we created a text file with 250 lines of insert commands. Each command inserted different data type. In test2, test3 and test4, we increased the inserts to 500, 750 and 1000 inserts, respectively. Each file is executed with a command at the accumulo terminal. Each successful insert returns a timestamp, which we use to compute the time between the first and last insert. This test aims to real-time inserts, such as data from onsite RFI monitoring. We find an average of 16 insert rate (inserts per second), which implies 960 inserts in a minute. However, as the number of inserts increases, the insert rate also increase, as shown in Figure 6b.

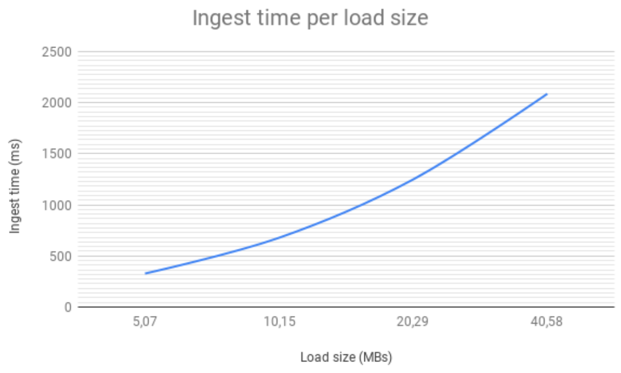
Table 4: Showing number of inserts per seconds

test	inserts	seconds	inserts/second
test1	250	15.79	15.83
test2	500	31.20	16.03
test3	750	45.86	16.35
test4	1000	60.92	16.41
average			16.15

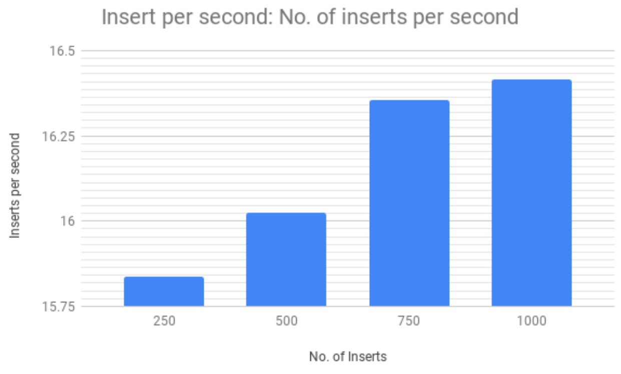
Similarly, we tested ingest speed for our prototype by loading measurement data of varying size into array stores. In test5, 5 MB of measurement data was modelled into a CSV file, which loads directly into an array schema. This was repeated in test6, test7 and test8 with file size doubling with each successive test i.e 5 MB, 10 MB, 20 MB and 40 MB (Table 5). On average, we observe that about 1GB data file can load in a minute, and as file size doubles, ingest time increases by a similar magnitude, thus load size varies proportionally with ingest time. This observation is in accordance with the inherent principles on which scalable databases have been built i.e to be able to scale with the load, unlike conventional model where the behaviour is expected to show a deterioration as the load ingested increases. In general for these databases, the ingest rate increases as the load size increases, as shown in Figure 6a.

Table 5: Showing ingest rate as load increases

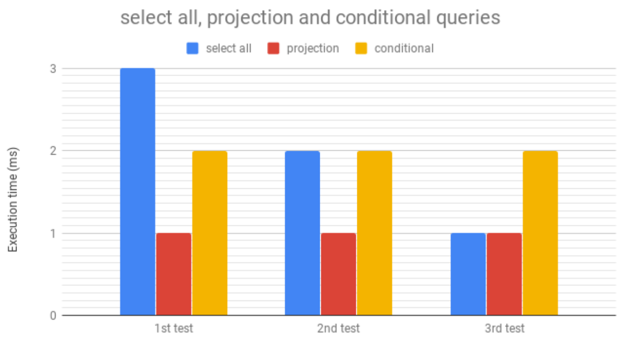
test	load size (MBs)	seconds	ingest rate (MBits/sec)
test5	5.07	0.328	15.46
test6	10.15	0.678	14.97
test7	20.29	1.249	16.24
test8	40.58	2.087	19.44
average			16.53



(a) Ingest speed



(b) Inserts per seconds



(c) Query execution time

Figure 6: Performance measurements on the database prototype.

Finally, Figure 6c graphs the query speeds for different categories of query i.e queries that select or scan all records, queries that project a specific table view and queries that return records while certain conditions hold. It is evident that the *selects all* command that scans the entire records requires longer execution time than the other two queries. However, on second and third test, the execution time drops, while it remains constant for the other categories. The reduction in execution time may be explained in form of memtables that are designed in distributed databases built on shared-nothing architecture. They provide temporary memory to hold data indexed for some time before being flushed out.

6 CONCLUSIONS

In order to address the challenges astronomers at MeerKAT encounter while trying to store and access RFI datasets, we have provided both conceptual and logical designs for a database based on the underlying principle of many databases, many data models. We have implemented a prototype of the database model into the Docker environment. Preliminary tests showing linearly scaling of data ingestion as data sizes increases and fast queries. These preliminary tests are limited to that of a single Linux Box loaded with Docker, however we plan to expand this to multiple machines or multiple cloud try to emulate a real-life data intensive environment. Further tests on our database model will aim to establish the number of users database can support and test for how easily a database that is distributed across a number of nodes either on cloud or servers can be updated. Finally, we find the polystore model to be a good choice for scalable datasets such as RFI data.

REFERENCES

[1] 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):180, apr 2018. doi: 10.3847/1538-4357/aab35a. URL https://doi.org/10.3847%2F1538-4357%2Faab35a.

[2] UN Task Team. The Big Data Revolution for Sustainable Development. *UN stats*, March 2017.

[3] R. D. Ekers and J. F. Bell. Radio Frequency Interference. In *The Universe at Low Radio Frequencies, IAU Symposium*, volume 199. IAU, February 2000.

[4] Independent Communications Authority of South Africa (ICASA). National Radio Frequency Plan 2013. Technical Report 36336, Republic of South Africa.

[5] G. Bianchi. Medicina Radio Astronomical Station, May 2016. URL http://www.med.ira.inaf.it.

[6] John M. Ford and Kaushal D. Buch. RFI mitigation techniques in radio astronomy. In *2014 IEEE Geoscience and Remote Sensing Symposium*, pages 231–234, July 2014. doi: 10.1109/IGARSS.2014.6946399.

[7] Fridman, P. A. and Baan, W. A. RFI mitigation methods in radio astronomy. *Astronomy & Astrophysics*, 378(1):327–344, 2001. doi: 10.1051/0004-6361:20011166. URL https://doi.org/10.1051/0004-6361:20011166.

[8] R. P. Millenaar and A. J. Otto. Innovations in instrumentation for RFI monitoring. In *2016 Radio Frequency Interference (RFI)*, pages 65–68, October 2016. doi: 10.1109/RFINT.2016.7833533.

[9] Prashant Kumar. An overview of architectures and techniques for integrated data systems (IDS) implementation. 2012.

[10] Kamalpreet Singh and Ravinder Kaur. Hadoop: Addressing challenges of Big Data. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 686–689, February 2014. doi: 10.1109/IAdCC.2014.6779407.

[11] Peter Buneman. Why Scientists Don't Use Databases. April 2002.

[12] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4):34–41, December 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107503. URL http://doi.acm.org/10.1145/1107499.1107503.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL http://doi.acm.org/10.1145/362384.362685.

[14] 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. *Proc. VLDB Endow.*, 2(2):1534–1537, August 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687584. URL https://doi.org/10.14778/1687553.1687584.

[15] R. Zafar, E. Yafi, M. F. Zuhairi, and H. Dao. Big Data: The NoSQL and RDBMS review. In *2016 International Conference on Information and Communication Technology (ICICTM)*, pages 120–126, May 2016. doi: 10.1109/ICICTM.2016.7890788.

[16] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. ISSN 0163-5808. doi: 10.1145/1978915.1978919. URL http://doi.acm.org/10.1145/1978915.1978919.

[17] Carlos Coronel and Steven Morris. *Database Systems: Design, Implementation, and Management.* Cengage Learning, 12 edition, 2016.

[18] T. Li, Y. Liu, Y. Tian, S. Shen, and W. Mao. A Storage Solution for Massive IoT Data Based on NoSQL. In *2012 IEEE International Conference on Green Computing and Communications*, pages 50–57, November 2012. doi: 10.1109/GreenCom.2012.18.

[19] K. Srivastava and N. Shekokar. A Polyglot Persistence approach for E-Commerce business model. In *2016 International Conference on Information Science (ICIS)*, pages 7–11, August 2016. doi: 10.1109/INFOSCI.2016.7845291.

[20] S. Prasad and S. B. Avinash. Application of polyglot persistence to enhance performance of the energy data management systems. In *2014 International Conference on Advances in Electronics Computers and Communications*, pages 1–6, October 2014. doi: 10.1109/ICAECC.2014.7002444.

[21] Zuohao She Adam Dziedzic Tim Mattson, Vijay Gadepally and Jeff Parkhurst. Demonstrating the BigDAWG Polystore. System for Ocean Metagenomic Analysis. In *Creative Commons Attribution*, 2017.

[22] 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. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, September 2016. doi: 10.1109/HPEC.2016.7761636.

[23] P. Chen, V. Gadepally, and M. Stonebraker. The BigDawg monitoring framework. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 9 2016. doi: 10.1109/HPEC.2016.7761642.

[24] Tim Kraska Sam Madden Tim Mattson Jennie Duggan, Aaron Elmore and Michael Stonebraker. The BigDawg Architecture and Reference Implementation. In *New England Database Day*, 2015.

[25] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science Engineering*, 15(3):54–62, May 2013. ISSN 1521-9615. doi: 10.1109/MCSE.2013.19.

[26] Wouter Buytaert Marius Appel, Florian Lahn and Edzer Pebesma. Open and scalable analytics of large Earth observation datasets: From scenes to multidimensional arrays using SciDB and GDAL. *ISPRS Journal of Photogrammetry and Remote Sensing*, 138:47–56, April 2018.

[27] Michael Stonebraker Gary Planthaber and James Frew. EarthDB: Scalable Analysis of MODIS Data Using SciDB. In *BigSpatial@SIGSPATIAL*, 2012.

[28] 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. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 8315–8318, August 2011. doi: 10.1109/IEMBS.2011.6092050.

[29] GEOTRACES. Geotraces.org, January 2017. URL http://www.geotraces.org.

[30] S. de Fatima Poppi Borba. Extending the UML for dimensional models in object-oriented database. In *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, pages 1150–1154, August 2005. doi: 10.1109/DEXA.2005.90.

[31] N. Naik. Docker container-based big data processing system in multiple clouds for everyone. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, October 2017. doi: 10.1109/SysEng.2017.8088294.

[32] W. Velásquez, A. Munoz-Arcentales and J. S. Rodriguez. A case study: Ingestion analysis of wsn data in databases using docker. In *2018 1st International Conference on Computer Applications Information Security (ICCAIS)*, pages 1–6, April 2018. doi: 10.1109/CAIS.2018.8441979.