Performance evaluation of an integrated RFI database for the MeerKAT/SKA radio telescope

Gerald Nathan Balekaki
University of Cape Town
Department of Computer Science
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

For radio telescopes, radio frequency interference from terrestrial and other sources is a recognized problem that contaminates the signal (RFI) and must be tracked and ultimately removed. At the MeerKAT/SKA telescope, RFI is recorded with a variety of devices, including telescopes, sensors, and scanners; but the combination of data from these multiple sources to yield a unified view of RFI remains a challenging problem. Previously, we demonstrated that a scalable database model with an implementation based on the Polystore framework is a potential solution for RFI monitoring. Here we extend this work, implementing the database model in an integrated environment and evaluating its performance across a range of workloads with three data stores: SciDB, PSQL, and Accumulo. We find that SciDB and Accumulo scale better than PSQL under multi-user environments. Results show a minimal latency as low as 0.02 seconds, irrespective of the location, and data store type. Further, integrated APIs provide single notation and are 5% faster than third-party APIs. Our findings thus provide a guide to the proposed integrated RFI system at MeerKAT/SKA radio telescope.

CCS CONCEPTS

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

KEYWORDS

RFI monitoring, diverse datasets, database integration, polystore database

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAICSIT '20, September 14–16, 2020, Cape Town, South Africa
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8847-4/20/09…$15.00
https://doi.org/10.1145/3410886.3410910

ACM Reference Format:
Gerald Nathan Balekaki, Michelle Kuttel, Anja Schroeder, Sarah Blyth, and Sonia Berman. 2020. Performance evaluation of an integrated RFI database for the MeerKAT/SKA radio telescope. In *Conference of the South African Institute of Computer Scientists and Information Technologists 2020 (SAICSIT '20), September 14–16, 2020, Cape Town, South Africa.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3410886.3410910

1 INTRODUCTION

The Square Kilometre Array (SKA) telescope project aims to construct the largest and most sensitive telescope in the world, with the first phase (about 10% of the planned full SKA) expected to complete in 2020. The SKA precursor, MeerKAT, is currently in operation in South Africa and comprises approximately 1% of the final SKA sensitivity [1].

In radio astronomy, radio frequency interference refers to any signal captured by a radio telescope that did not originate from the observed target in the sky. Sources of RFI range from human-generated to natural objects such as the sun. With the expansion of technology, RFI contamination of the radio signal is on the rise. RFI signals are typically stronger than the weak celestial signals of interest and therefore have a dramatic deleterious effect on observation data [2]. Radio astronomers therefore collect and store RFI data, in order to explore and understand the nature of the RFI sources, with a view to determining appropriate mitigation approaches.

The RFI data collect is dynamic and diverse, including images, text, documents, arrays and tables. The data sets also tend to vary widely in quality, coverage, accuracy, and period [3], making data storage a concern. For example, at the MeerKAT/SKA telescope, RFI data is collected using a radio telescope as well as several monitoring devices including fixed and mobile stations, sensors, and scanners [4]. The use of several devices introduces incompatibilities in data formats, models, languages, and infrastructure, which exacerbates the disintegration gap [5]. Further, the isolated data environment at MeerKAT, as well as other international radio telescopes [6], hinders fast access and storage due to poor integration.

Traditional flat file formats, such as CSV, are commonly used by many scientists for their data storage. However, they lack the structure necessary for indexing multiple files. More structured file formats, such as HDF5 [7] and NetCDF [8], were introduced to deal

with large and diverse scientific data, but are still built on the basic principle of a flat file [9]. A database management solution (DBMS) is preferable, but the biggest challenge in using databases to store scientific data lies in integrating a variety of data elements. Traditionally, a distributed database approach has been a solution for disintegrated data environments. The SciDB platform supports fast searches, bulk loading, and ad-hoc analysis and is used in EarthDB to store Moderate Resolution Imaging Spectroradiometer data on earth dynamics and processes [10]. Studies on science-oriented DBMSs indicated that the PostgreSQL (PSQL) database system [11] is a suitable solution for astronomy due to its extensibility and powerful features. However, a distributed database approach does not provide the interactivity and autonomy necessary for an on-site RFI monitoring environment.

An alternative is a heterogeneous database that integrates many database systems while maintaining application-specific properties, integrity control, and safety control [12]. Each database system requires independent operation environments, database engines, data structures, and semantics; and must also maintain the accuracy of a query during schema translation as data moves from one system to another. Data integration aims to provide a global view of the entire data environment and make storage faster and less cumbersome. In previous work, we designed a scalable database model[5] based on a polystore framework [13, 14]. Our database model uses integrated middleware that coordinates multiple data models, languages, and engines, to facilitate uniform modality and query notation. Our design philosophy lies in leaving data in its native format, as we model the data to fit in the suitable store. This means distinct data structures dictate their store technologies and calls for flexible logical and physical designs that can accommodate several data structures.

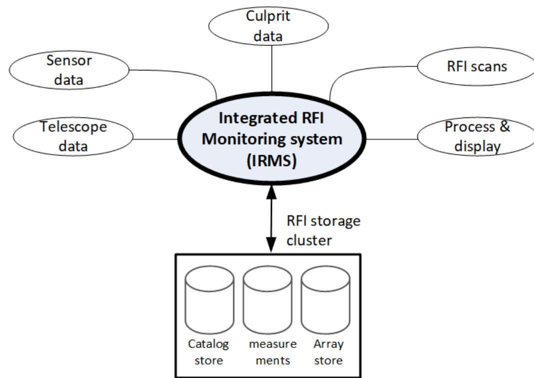

Figure 1: The proposed integrated RFI monitoring system (Source: SKA RFI Working Group)

Our scalable database model, illustrated in Figure 1, incorporates an integrated RFI monitoring system linked to an RFI storage cluster of autonomous data stores. The storage cluster consists of three data stores: a Postgres (PSQL) [11] store for structured relational data, an Accumulo [15] store for unstructured key-value data and a SciDB [9] store for arrays. Our relational data includes RFI metadata and system (receiver and transmitter) information; key-value data includes RFI measurement data, text in RFI reports and RFI scans and multi-dimensional data, such as 3D frequency, time, and polarisation data, is stored in arrays.

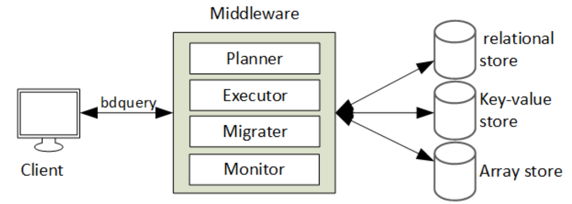

Figure 2: Basic polystore database framework [16]

Figure 2 illustrates a basic polystore framework with an integrated middleware that coordinates the three database stores appropriate for our proposed integrated environment. The middleware has four components that coordinate operations across engines; the planner, executor, migrator, and monitor [14]. This middleware enables several new database operations, including islands, shims, and casts. Islands provide users with a variety of programming and data models. Cast operations facilitate the migration of data between engines, and shim operations facilitate translation of data from one data format model to another.

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 utilise just the array database, while a join operation between time-series data and catalog data may access both the array and relational databases. 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 [17].

The polystore database model is a relatively new and promising model for isolated data environments, but its performance has not been clearly demonstrated. In this work, we assess the performance of our model under a monitoring data-intensive environment. Our queries are built to extract data from multiple stores with single or multiple users. The query results in this work are limited to 100% read-only workload, which is one of the Yahoo Cloud Serving Benchmark workload standards used to evaluate scalable systems [18, 19]. We expect that integrating RFI data in our model will not only provide uniform storage, but will also enhance database performance by reducing data transfer delays associated with accessing separate data stores.

2 METHODOLOGY

We implement the evaluation setup in a dockerized environment. Docker [20] is an application that creates a virtualized environment suitable for running multiple applications that suffer from integration and interoperability complexities [21]. The setup is typical of our monitoring environment that involves RFI detection, deep analytics, and storage applications. We install the polystore implementation framework using a bigdawg prototype available

on GitHub repository [22]. We use Docker Engine (*version 19.03.5, build 633a0ea*) to host the database setup, which contains three database applications, that is, PSQL, SciDB, Accumulo. The host machine runs Ubuntu Operating systems, 16GB RAM, and 1TB disk capacity. The setup consists of a local clients and a remote database server. The client can access the database through an integrated API using an HTTP request shown in Figure 3.

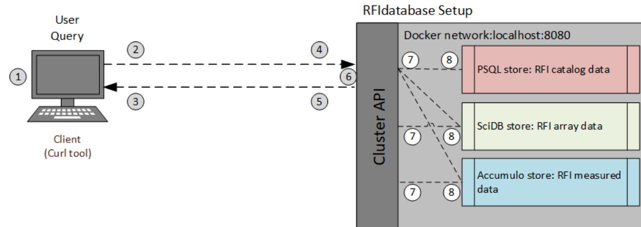

Figure 3: Evaluation setup for the polystore implementation of the integrated RFI database [23]. The measured parameters are as follows: (1) response time; (2) upload size (3) download size; (4) upload speed; (5) download speed; (6) connection time; (7) namelookup time; (8) latency.

We created a database by implementing a logical and physical data model, and prepared data using CSV files to enable faster loading of bulk files. The largest sample size considered is about 500MBs, mostly of an array type. Most RFI data is measurement data that can fit well in a 2D or 3D structure. We run experiments by designing sample queries that emulate our monitoring environment, as we measure performance parameters (listed in Table 1 and labeled in Figure 3). To evaluate the performance of the database, we focussed on download speed in kB/sec, the response time in seconds, and latency in seconds. Download speed and response time are measured to establish how fast to download RFI data from the database. We measure latency to determine delay associated with each data store. Connection and namelookup times are measured to determine the source of latency. We also measure database throughput to determine the stability of the database cluster.

Table 1: Test parameters for the database implementation.

label	parameter	description
1	response time	total time for the query transaction
2	upload size	size of data uploaded
3	download size	size of data downloaded
4	upload speed	average speed for an upload
5	download speed	average speed for a download
6	connection time	time from start until TCP connection is complete
7	namelookup time	time from start until name resolving is complete
8.	latency	time taken before the actual data transfer begins.

The query response time is the time taken for a full query transaction to complete. Upload sizes and speed are data sizes and speed measured upon successful upload. We measure upload and download parameters after successful upload and download operations. Latency is the time measured before the actual data transfer (pre-transfer time) begins, which includes connection time and name lookup time (connection and namelookup time are therefore components of the latency). Connection time is the time taken for a database connection to complete, while namelookup time measures the time to resolve a name within a database. Since latency is dependent per store. It is important to establish a portion of latency per store. We also compute connection and name lookup as percentages of latency, to determine the biggest source of delay. If most of the delay is attributable to connection time, then a high-quality network connection can be recommended.

2.1 Sample Queries

We measure performance by running user queries that return data residing in one or more data stores. We categorise the queries as simple, complex, aggregate, and join. A simple query fetches data from in a single store, while a complex query fetches data from more than one store. An aggregate query fetches and computes and returns a resultant value, while a join query consists of a join operation that coordinates related data from two or more data objects. An example of the syntax of a simple query is as follows.

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

The client uses a client URL (cURL tool) to post data in the form of a URL onto the database server (192.168.0.117) via HTTP port 8080. The database uses the information stored in the islands to direct the query to respective engines. In this case, the text island is linked directly to the Accumulo (key-value/text) store engine that scans the transmitter data object. RFI related data is arranged chronologically in columns using unique identifiers (rfi001 and rfi004). The query will return all RFI data with keys between rfi001 and rfi004 with their associated values, and also compute the corresponding value of the total time to complete the entire query transaction.

Our queries emulate a data-intensive environment that involves bulk retrievals and many users with concurrent access. We linearly increased the number of records from 2000 up until 16000. We vary the number of users from 2 until 12. Each user reads data of not less than 1MBs in size from the database. Our queries are limited to Read-only workloads. This means the workloads in this work involves 100% reading records from the database. This is one of the YCSB benchmarks we applied to evaluate our scalable database model. Others require to configure workloads to consists about 50% reads and 50% updates; 95% of reads and 5% of updates; 95% of reads and 5% of inserts; and Read-modify-write – where the query will read a record, modify and write back the changes [19].

2.2 Test Environments

We consider three test environments that characterise our monitoring environment. The first environment relates to a linear data growth pattern where data sizes and the number of records increases linearly. This growth pattern is ideal for any organisation. We also model a multi-user environment where we increase the number of database users linearly, with a workload of about 1 MB per user. Here, we increase the workload by increasing the number of users, whereas for the linear test where we varied the number of records. The second environment concerns measuring bulk ingestion. This is a key factor in our monitoring environment that requires fast loading of bulk datasets into the database, a common requirement for modern databases [24]. To measure load speed, a

data file of 1.2 MB is loaded into each of the three stores to measure average load speed and total time taken. It is important to note that each store has a unique loading technique dependent on its level of development. Lastly, we consider an API environment. Here we compare the query performance of native API with that of a RESTful third-party API – *Insomnia* [25] – that applies to our setup. Native API is inherent in the polystore model, while the third party is an added application that runs on top of the RFI database.

3 RESULTS

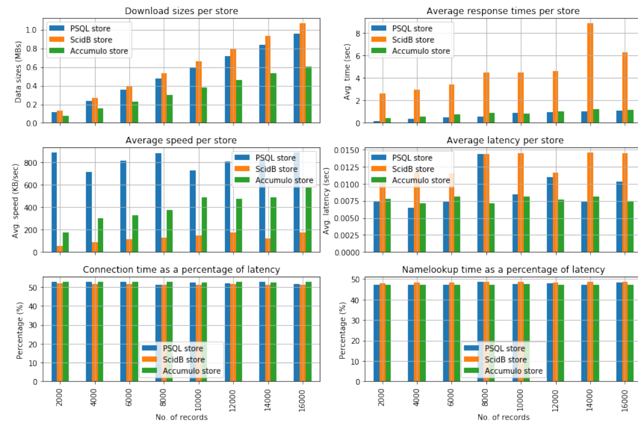

Figure 4: Individual store performance as the number of records scales linearly. Download speed, response time, and latency are computed as averages. Connection and namelookup time are represented as percentages of latency.

Figure 4 shows the performance of individual stores as the number of records and data sizes increase linearly. This allows us to assess store performance in terms of download speed, average response time, download sizes, and latency.Note that the PSQL and SciDB stores download significantly larger data sizes than Accumulo, given the same number of records. The difference in data sizes is attributed to the varying data structures in each store: the Accumulo data structure holds a single value per record; whereas PSQL and SciDB that hold multiple values per record.

On the other hand, the average response time in SciDB is significantly higher than that of Accumulo and PSQL. This is because of the SciDB structure that stores much more RFI data per record than Accumulo and PSQL. About 40% of the sample dataset is structured as arrays and therefore stored in the SciDB store, leaving Accumulo and PSQL share 30% each. As we increase the number of records, the SciDB response time increases. These higher response times affect download speeds adversely: SciDB has a much lower speed as a result of longer response times, and subsequently longer transfer times. Any query that fetches data from the SciDB store will be slowed down.

The average latency in all data stores is less than 0.02 seconds. We observer the latency for PSQL and Accumulo drops as low as 0.01 seconds. High latency in SciDB is a result of longer response times and longer data transfers. Databases that experience average latency ranging from 0.01 and 0.02 seconds are recommended for handling large datasets. We observe that connection time contributes just over 50% of total latency and namelookup slightly below 50%. There is less congestion from single-user connections, therefore, the latency is fairly distributed between database connections and namelookup times.

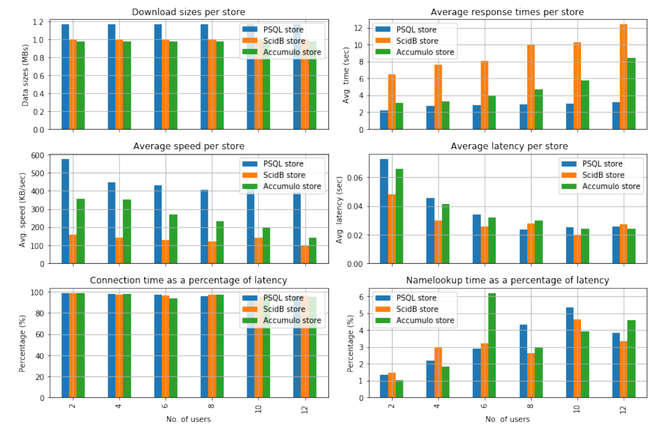

Figure 5: Individual store performance in a multi-user environment. Results show average response time, download speed, and latency. Connection and namelookup time as percentages of latency.

Figure 5 shows the store performance as the number of users increases. Download sizes for all stores are relatively similar simply because the same size of workload (about 1 MB) was considered. Despite a similar size in workload, we still observe differences in download sizes because of the difference in structure per store. SciDB data store registers the highest average response time of the three stores. Accumulo and PSQL stores have a much lower response time, but PSQL with the lowest. Average response time directly affects average speeds. This can be observed between response and speed plot where SciDB and Accumulo high response results in lower speeds.

Our latency results reduce with an increase in users for all stores. Here, the average latency is slightly over 0.02 seconds. PSQL and Accumulo, record higher latency than SciDB. This indicates that PSQL and Accumulo are much impacted than SciDB when more users access the database at the same time. The longer latency in PSQL is attributed to its lengthy connection process required to access the database. We also observe that most latency (over 98%) is attributable to connection time, and 2% to namelookup time. This is expected as multiple users try to connect and access the remote server that hosts the database. Long connection times are a result of several connections to the database and likely to cause unnecessary re-transmissions or completely lost transmissions (see Figure 6, network variability performance).

Figure 6 shows performance variation due to network variability in a multi-user environment. The results show performance variation in terms of database throughput and query completion time. This result demonstrates how the database performance is disrupted by numerous network variations typical of multiple-user

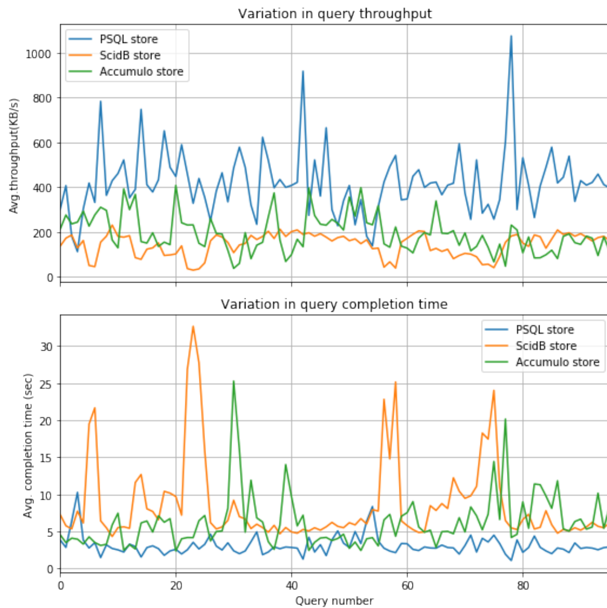

Figure 6: Performance variation in throughput and completion time due to network variability.

environments. We observe a higher throughput in PSQL and Accumulo, despite a relatively high variation with time. This high variation is attributed to the store's ability to accommodate many users, while the high throughput is as a result of shorter response and transfer times. Unlike SciDB, that show low throughput, but with a relatively stable impact. This indicates that SciDB performance is less disrupted as more users access the database cluster. This is in line with the underlying design principles of Accumulo and SciDB that guarantees stability under unstable data environments that are filled with many failovers, errors, and crashes. The variability results can improve with high-quality network speeds that allocate enough bandwidth to multiple users to reduce competition for network resources.

Table 2: showing performance of loading technique per store

item	size (MB)	time (sec)	avg.speed (KB/sec)
PSQL	1.20	0.16	7.30
SciDB	1.20	0.12	9.80
Accumulo	1.20	324.30	0.004

Table 2 demonstrates results obtained from a loading test. This test is crucial to our environment, which is characterised by frequent bulk loading. A workload of 1.2 MB was loaded in three separate stores as we measured the time it takes to load. The results show that SciDB loads much faster with the highest speed of 9.8 MB/sec and the shortest loading time of the three stores, followed by PSQL with a speed of 7.3 MB/sec. Accumulo has the longest loading time that greatly affects the speeds. This fastest loading speeds in SciDB and PSQL are attributed to the more advanced loading techniques found in each of the stores. Both stores are built on a foundation of a traditional relational database that has existed for close to 5 decades. On the other hand, poor loading speeds found in Accumulo is as a result of a relatively new data model that uses basic ETL (extract, transform, and load) techniques for loading bulk data into the database.

Table 3: showing performance of native vs. third-party API

query	native (sec)	3rd party API (sec)	% increase
simple	0.57	0.61	6.59
complex	0.68	0.72	5.48
aggregate	0.59	0.59	0.04
join	1.34	1.41	4.94

Table 3 shows indicate results obtained from two API environments, that is native and a thirty-party API. These test results are vital to determine how the model performance in different API environments. API is a key feature when dealing with multiple and isolated data stores. We ran the same type of query (either simple, complex, aggregate, or join) to return similar data from two different APIs, as we measured average response time.

Table 3 indicate third-party API with a higher response time than native API. On average, we observed a significant percentage increase in response time of over 5% in third-party API, except for aggregate queries that show an insignificant increase of about 0.04%. This is because aggregate queries are concerned with a single aggregated value, and they are not affected by the nature of the API. In other words, aggregated query operations do not affect the average response time irrespective of the API used to fetch the data. Overall, the native API performance results for all query types outperform those of a third-party API by about 5% in response time.

3.1 Limitations

This work is limited to a workload that consists of 100% reading from the database. Considering other workload variations may give a deeper understanding of the performance of the database model. We limited the sample data size to about 500MBs file due to performance issues associated with the limited specifications of the Host machine of 16 of RAM and 1TB of capacity. A machine with higher specifications is likely to improve on speed reading data from the database.

4 CONCLUSIONS

In this work, we conduct performance tests of our polystore model for RFI under different test environments. We demonstrate that database queries encounter a latency of fewer than 0.02 seconds this facilitates faster transfers, though more work can be done to reduce latency further e.g. better network connections or newer optimizing techniques. PSQL and SciDB possess more advanced and faster loading scripts. The native API performance results outperforms those of a third-party API by about 5%. SciDB throughput is more reliable under unstable network connections, and about 98% of latency is a result of poor database connections Our results show that the polystore model can be a solution for disintegrated data environments. This work will assist scientists and astronomers to

smoothly implement and assess the RFI database at SKA/MeerKAT radio telescope.

ACKNOWLEDGMENTS

This work was financially supported by Hasso Plattner Institute for Digital Engineering through the HPI Research School at UCT. The authors also wish to acknowledge SARAO and SAAO for the collaboration. Special thanks to the RFI SKA Working group of South Africa.

REFERENCES

[1] Jonas, J. A. et al. 2018. The MeerKAT Radio Telescope. In *2016 MeerKAT Science: On the Pathway to the SKA (MeerKAT2016).* https://doi.org/10.22323/1.277.0001

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

[3] Dong, Xin and Srivastava, Divesh. 2013. Big Data Integration. *Proceedings of the VLDB Endowment* 6, 1245–1248. https://doi.org/10.1109/ICDE.2013.6544914

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

[5] 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 *Proceedings of the South African Institute of Computer Scientists and Information Technologists 2019* (Skukuza, South Africa) *(SAICSIT '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 8 pages. https://doi.org/10.1145/3351108.3351127

[6] Prashant Kumar. 2012. An overview of architectures and techniques for integrated data systems (IDS) implementation. *Integrating Factors Inc* (2012).

[7] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An Overview of the HDF5 Technology Suite and Its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (Uppsala, Sweden) *(AD '11)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/1966895.1966900

[8] R. Rew and G. Davis. 1990. NetCDF: an interface for scientific data access. *IEEE Computer Graphics and Applications* 10, 4 (1990), 76–82.

[9] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. 2013. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science Engineering* 15, 3 (05 2013), 54–62. https://doi.org/10.1109/MCSE.2013.19

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

[11] Bo Han, Yan-Xia Zhang, Shou-Bo Zhong, and Yong-Heng Zhao. 2016. Astronomical data fusion tool based on postgresql. *Research in Astronomy and Astrophysics* 16, 11 (2016), 178.

[12] Cao Jie, Hou Wen, and Cai Tingyou. 2008. Research of Heterogeneous Database Integration system based on E-business. In *2008 IEEE International Conference on Service Operations and Logistics, and Informatics*, Vol. 1. 186–189.

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

[14] Mattson Tim, Gadepally Vijay, She Zuohao, Dziedzic Adam, and Parkhurst Jeff. 2017. Demonstrating the BigDAWG Polystore. System for Ocean Metagenomic Analysis. In *Proceedings of Conference on Innovative Data Systems Research (CIDR17)*.

[15] M. Wall, A. Cordova, and B. Rinaldi. 2013. Accumulo Application Development, Tables Designs, and Best Practices. In *O'Reilly*.

[16] Gadepally V, Chen P, Duggan J, Elmore A, Haynes B, Kepner J, Madden S, Mattson T, and Stonebraker M. 2016. The BigDAWG polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2016.7761636

[17] BigDAWG Developers. 2017. *BigDAWG Documentation, Release 0.1.* Technical Report. Intel Science and Technology Center(ISTC).

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

[19] M. Barata, J. Bernardino, and P. Furtado. 2014. YCSB and TPC-H: Big Data and Decision Support Benchmarks. In *2014 IEEE International Congress on Big Data*. 800–801.

[20] Babak Bashari Rad, Harrison Bhatti, and Mohammad Ahmadi. 2017. An Introduction to Docker and Analysis of its Performance. *IJCSNS International Journal of Computer Science and Network Security* 173 (03 2017), 8.

[21] A. Munoz–Arcentales W. Velásquez and J. S. Rodriguez. 2018. A Case Study: Ingestion Analysis of WSN Data in Databases using Docker. In *2018 1st International Conference on Computer Applications Information Security (ICCAIS)*. 1–6. https://doi.org/10.1109/CAIS.2018.8441979

[22] BigDAWG (Big DAta Working Group). [n.d.]. *BigDAWG Polystore.* Retrieved March 10, 2020 from https://github.com/bigdawg-istc/bigdawg.git

[23] Jennie M Duggan, Aaron J. Elmore, Tim Kraska, Sam Madden, Tim Mattson, and Michael Stonebraker. 2015. The BigDawg Architecture and Reference Implementation. http://db.csail.mit.edu/nedbday15/ Eighth Annual New England Database Day, NEDB Day ; Conference date: 30-01-2015 Through 30-01-2015.

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

[25] Insomnia. [n.d.]. *Design and debug APIs like a human, not a robot.* https://insomnia.rest/