SCALABLE TIME SERIES DOCUMENTS STORE

M.J. Slabber*, F.J. Joubert, M.T. Ockards
SKA SA, Cape Town, South Africa

Abstract

Data indexed by time is continuously collected from instruments, environment and users. Samples are recorded from sensors or software components at specific times, starting as simple numbers and increasing in complexity as associated values accrue e.g. status and acquisition times. A sample is more than a triple and evolves into a document. Besides variance, the volume and veracity also increase and the time series database (TSDB) has to process hundreds of GB/day. Also, users performing analyses have ever increasing demands e.g. in <10s plot all target coordinates over 24h of 64 radio telescope dishes, recorded at 1 Hz. Besides the many short term queries, trend analyses over long periods and in depth enquiries by specialists around past events e.g. critical hardware failure or scientific discovery, are performed. This paper discusses the solution used for the MeerKAT radio telescope under construction by SKA SA in South Africa. System architecture and performance characteristics of the developed TSDB are explained. We demonstrate how we broke the mould of using general purpose database technologies to build a TSDB by rather utilising technologies employed in distributed file storage.

OVERVIEW

This paper describes the updated Katstore [1], the storage system developed to store the values, status and other information about sensors in the MeerKAT [2] Control And Monitoring (CAM) system [3]. Sensors and the CAM system are described in more depth in the Data Acquisition section.

Software components in CAM will send data points (samples) to Katstore to save and make available for analysis. All the samples received by Katstore are keyed on time and sensor name. This makes Katstore a time series database (TSDB); it is purposely built to have a fixed index on time. The data in Katstore is immutable and only grows over time, no update on a sample is allowed. It can be seen as an append-only database and samples do not need to arrive in chronological order.

For flexibility and ease of implementation, the samples are packed as JavaScript Object Notation (JSON) [4] objects by the software components that collected the samples. Any valid JSON is accepted. Katstore only requires that each sample contains the keys *name* and *time*, where *name* is the sensor name and *time* is the time in Coordinated Universal Time (UTC). The sample structure is discussed in the Samples section. These semi-structured samples, as JSON objects, are generally referred to as a document in database systems. Document storage has been popularised in recent years with the NoSQL movement, with document orientated databases such as CouchDB, Elasticsearch, and MongoDB. Making each sample a document removes the need for application knowledge in Katstore and future-proofs the implementation. New fields can be added and removed without requiring changes to Katstore and there is no fixed schema for a sensor sample.

The software components that collect the samples will publish the samples to the message bus [5] at intervals that can be configured per sensor. Katstore will subscribe to the per sensor archive subject on the message bus and store the published samples. The samples are first written to a buffer and will at a later stage be written to the archive. This buffer and the archive system are described in more detail in sections Samples Buffer and Samples Archive respectively.

Katstore has a query interface that supports several ways to access the stored samples. This is discussed in more detail in the section Query Interface. Samples are fetched from the buffer and/or archive when the user queries the system; this is done transparently to the user.

In the Performance section we describe the initial performance evaluation of the Katstore system.

DATA ACQUISITION

The detail of the MeerKAT CAM system was well described in 2015 by Marais [3] and the sensor sample data acquisition was illustrated in the same year by Slabber et al. [6]. Since then some changes to where sensor samples are collected from and how samples are transported have been implemented, these changes are explained by Joubert et al. [5].

An abridgement of that work is given here for completeness. MeerKAT CAM has many software components some components connect to hardware devices and others connect to software components. All inter-component communication is done with Karoo Array Telescope Communication Protocol (KATCP) [7]. Components can call requests on connected components for control purposes. A KATCP request is analogous to method or command calls of other platforms. For monitoring purposes KATCP, provides the concept of sensors. For the purpose of archiving the components that make up the MeerKAT CAM system publish sensor samples to different subjects on the message bus, the publish rate is controlled by the system configuration. Katstore subscribes to the archive subjects and store the samples to the buffer.

Sensors

A sensor is a fundamental concept in KATCP [7] and a rich collection of sensor types are available. The following types are currently supported *integer*, *float*, *boolean*, *timestamp*, *discrete*, *address* and *string*. Sensors always have

* martin@ska.ac.za

```
{
 "name": "m000_rsc_rxl_cryostat_pressure",
 "time": 1505982067.202219,
 "value": 1013.25,
 "status": "nominal",
 "value_ts": 1505977839.44
}
```

Table 1: Example sensor sample

a status and the following statuses are supported *unknown*, *nominal*, *warn*, *error*, *failure*, *unreachable* and *inactive*.

In KATCP sensor sampling is performed by the server based on a sampling strategy provided by the client, this allow every connection to set up a unique sampling strategy. There are several sampling strategies available ranging from a fixed time interval (*period*) to on value change (*event*).

To facilitate the flexibility KATCP provide to the software components we had to ensure that the Time Series DataBase (TSDB) we built is flexible and compatible with at least all the scenario's supported in KATCP. Two of the primary features that lack in other time series databases are the correct handling of non numeric types of values and storage of associated status with each sensor value.

Samples

The structure of a sensor sample has previously [6] been explained in detail, but for the updated version of Katstore the fields in a sample are not critical. It must be mentioned that these fields are needed by applications that use the samples for analysis. Katstore guarantees that all the fields and values in the published samples are made available to applications and users.

Samples published on the message bus have a standard format [5], these samples are JSON objects and always have *time* and *name* fields, Table 1 shows an example sample.

The value of *time* can be a floating point value or a string value. When it is a floating point it is the time in seconds since the epoch of 1 January 1970 00:00:00 UTC, Unix time. When *time* is a string it has to be RFC3339 [8] compliant. The value of the *time* field will be replaced by the floating point representation of the supplied time.

The value of the *name* field is the normalised KATCP name, where all non-alphanumeric characters have been replaced by the "_" character. It is not necessary for Katstore to have *name* in any specific format since the *name* field is stored in UTF-8 and can be over 1024 characters long. But it was found that for usability sake it was best to normalise the storage sensor names and the sensor name used in queries.

Sensor Attributes

With each sensor there are several associated facts. We generally refer to these as sensor attributes or meta data. Sensors from KATCP always have the attributes description, type, unit, and parameters.

When a new sensor is created on a component, most often only on startup, the sensor attributes are published to a subject on the message bus. Katstore subscribes to attributes subjects on the message bus and will store the sensor attributes into a table in the database.

Attributes are published as JSON objects and the *name* field is the only prerequisite. Katstore has no limitation on the attributes that can be associated with a sensor, as long as it is JSON compliant. Certain attributes will improve the user experience when using the query interface.

Internal to Katstore the sensor information is stored in the **sensors_meta** table, with a row for each sensor. The attributes are stored in a column of type JSONB. JSONB is a JSON compliant data type in PostgreSQL [9].

A few extra fields are added when the attributes are published to the message bus. The KATCP name, as *katcp_name*, is added to the sensor attributes. This field is also used to build up a bidirectional parent-child hierarchy based on a "." in the KATCP name. Although not enforced by the KATCP specification, it is common for sensors to be named in such a hierarchical manner. The component name is added to the sensor attributes.

In addition to the given attributes of a sensor, Katstore keeps track of the last time the attributes were updated and the approximate first and last sample received. The first and last sample update times are approximations only, it is not updated on every sample processed.

SAMPLE BUFFER

Software components collect sensor samples and publish these to subjects on the message bus. The Bus2Db processes of Katstore subscribe to subjects on the message bus and store the samples into the database.

The components publish to an archive subject per sensor; the name of these subjects are made up of four parts (tokens). The first and second tokens are always `sensor` and `archive`. The component name and sensor name are the last tokens. The message bus system allows the Bus2Db processes to subscribe to all of these subjects in a simple manner by using a wildcard subscription, e.g. `sensor.archive.>`.

The Bus2Db processes all subscribe with the same queue name, which informs the message bus to only deliver a message to one of the Bus2Db processes. This ensures fair distribution of work among the Bus2Db processes.

The Bus2Db processes are written in Python3 [10] and heavily use the asynchronous capabilities of the Python3 built in asyncio library. The asyncio driver for both NATS and PostgreSQL are used. As Bus2Db receives messages it extracts the name and reads the time from the sample. The name, time and sample are written to the **samples_buffer** database table in batches using the Structured Query Language (SQL) command COPY FROM. Each sample is stored as a separate row in the **samples_buffer** table. Along with the name, time and sample columns, there is also an archived column in the **sample_buffer** table. This column holds a boolean value that is set to true once the sample has been archived.

Samples are stored in the buffer for a fixed duration. This samples-age is a configurable item and system administrators can set the value to make the best use of the system hardware. This value can be dynamically adjusted while the system is running. Samples are trimmed from the buffer in a very effective manner and only whole chunks where all samples in them have expired are deleted, which is more efficient than deleting individual rows.

PostgreSQL

PostgreSQL [9] has been a critical part of the Katstore system since the first incarnation. The use of the database has changed over time; initially it was only used to store the references to files for fast lookup and later to hold samples temporarily before they got written to file. In the latest version of Katstore we did away with the distributed in memory samples buffer [1] and use a Solid-State Drive (SSD) backed central database for storing the sensor samples (**sample_buffer** table). A few factors have contributed to the change in the design; the availability of fast storage, improved parallelism in later versions of PostgreSQL and the availability of a special time series extension, TimescaleDB [11] for PostgreSQL. The **sample_buffer** table is a TimescaleDB hypertable.

The JSONB data type recently added to PostgreSQL has made developing a scalable future proof storage system fairly simple. By selecting JSONB as the column type for samples and attributes we immediately simplify the processing, the samples and attributes already arrive as JSON, and we do not constrain the content of samples and attributes. By using JSONB we were able to simplify our implementation and at the same time allow for flexibility. Katstore can in the future be used to store time series samples that have a different sample construction to what is used with MeerKAT. The JSONB data type results in slightly more space being used per row, but this overhead has not proven to be of any significance.

TimescaleDB

TimescaleDB is a fairly new extension for PostgreSQL, it was purposely developed to improve the storage and processing of time domain data. It functions very similarly to our old implementation [1] but is written in C with an emphasis on performance. Using TimescaleDB was strait forward and allowed us to remove huge parts of application code dedicated to working with time series data.

But most importantly it gave us significantly better performance and scalability.

SAMPLES ARCHIVE

One of the factors to take into account when developing a very large TSDB is that very few of the samples stored will have frequent access. It could even be possible that some of the samples will never be accessed, but we do not know that up front. Thus to build a sufficiently large data warehouse in the traditional way where we have all the samples readily available for that off chance that it will be queried will incur a tremendous cost.

The approach in Katstore is to have a short-term buffer from where recent samples can be retrieved very quickly and to store all other (older) samples in a slower archive. We have chosen Ceph [12], a distributed storage system, as the platform for this archive. The archive is represented in the database as yet another table and standard SQL queries can be performed on this table. It is even possible to do JOIN operations on this table, but because of the data volume this is not advised. The table, **samples_archive**, is a federated table to the Ceph storage system and is implemented as a Foreign Data Wrapper (FDW) in Python3 using the Multicorn [13] library.

SQL modify operations (INSERT, UPDATE, and DELETE) are not supported on the **samples_archive** table. A second federated table also implemented as a FDW in Python3 using the Multicorn library was created for adding data. This table named **archiver** only supports INSERT operations.

The copying of samples from **samples_buffer** table to Ceph is done by calling a stored procedure in the database. This stored procedure takes in *name* as the only parameter, where name is the normalised sensor name. It will then take the oldest 10000 samples not yet archived for the sensor with the given name and perform an INSERT on the **archiver** table for each distinct day in the selected samples. All samples for a day are batched and inserted as one operation. The samples are flagged as archived. The stored procedure runs as a transaction and any failure will result in the archived flag not being set. Thus samples are inserted in batches, a JSON array of samples, into the **archiver** table but accessed as sample per row from the **samples_archive** table.

Ceph

Ceph [14] is a distributed fault tolerant storage system. Ceph allows us to combine many Hard Disk Drive (HDD) on many servers into one large storage system. Objects in the storage system are replicated and the failure of a drive or a complete server will not inhibit the system.

Ceph consists out of several components and several interfaces. It provides a filesystem like interface (CephFS), a block device interface (RBD), and an object storage interface (Rados). The filesystem interface and the block device interfaces are built upon the object storage interface. Katstore will use the object storage interface directly.

The Ceph cluster can easily grow as new hardware is added, and the systematic replacement of storage nodes with newer higher density ones can easily be performed.

By using Ceph for the archive storage, we are able to scale Katstore to several Petabytes without having to maintain an enormous database cluster. The Katstore usage patterns fits well with Ceph's storage design.

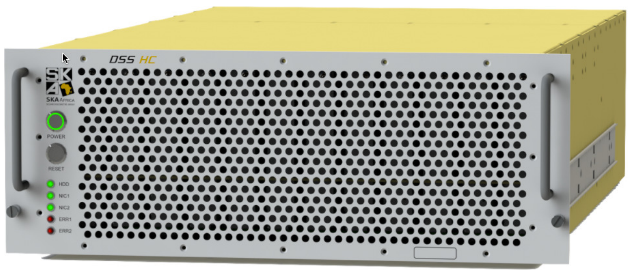

Figure 1: An SKA SA Storage Pod

The CAM deployment using virtual machines [15] is using Ceph as the storage layer for all the containers; production and development environments. The Science Data Processing (SDP) subsystem of MeerKAT has also chosen Ceph as the technology for storing the science data products of the telescope.

The SDP subsystem is in the process of deploying a Ceph system with a total raw capacity of 21 PB. This Ceph deployment will consist out of 55 storage pods, with 5 storage pods located at the telescope in the Karoo and the remaining 50 pods in the Centre for High Performance Computing (CHPC) in Cape Town. Katstore will use these clusters once they have been commissioned.

Storage Pods

It is worth mentioning that the hardware used for the Ceph clusters were specially developed by SKA SA. After thorough research it was found that no suitable product existed on the market and that at the required volume it would be feasible to develop such units. Table 2 lists the hardware specifications of a storage pod and Figure 1 shows one of the storage pods.

MeerKAT Katstore will archive to the on site Ceph cluster and objects will later be synchronised to the cluster at the CHPC for off-line analysis and long-term storage.

Table 2: Storage Pod hardware specifications

Processor (CPU)	Xeon 4 core 3.7 GHz
Memory (RAM)	64 GB
Network Interface	25 GbE
OS Disk	1 x 120 GB SSD
Storage Disk	48 x 8 TB HDD
Journal Disk	2 x 512 GB NVMe SSD

Rados Object Structure

Rados allows applications to create a named object and perform read, write, and append operations on that object; in a similar manner to doing those operations on an open file handle. Rados also supports the storage of key-value pairs with an object, Rados takes care of the storage and management of these additional key-value pairs.

No constraint is placed on the content of an object, Rados treats the contents as binary data.

We decided to create an object in Rados for each sensor per day, thus everyday a new object is created for each sensor. This created good logical separation between objects and fitted well with the historic queries.

The object name is constructed from the integer day and the normalised sensor name. An integer day is the number of days since the Unix epoch, 1 January 1970.

Such a coded object name allows the FDW to quickly get to the correct objects that will fulfil a query.

The contents of the objects are compressed and each batch of samples being archived are binary packed using Concise Binary Object Representation (CBOR) [16]. This packed byte array is compressed with the Zstandard [17] compression algorithm using the Blosc [18] library.

This compressed byte array and a small header make up a frame, frames are appended to the end of the Rados object.

The header is fixed length (11 bytes), it consists out of a preamble, a control, and a size field. The preamble is 2 bytes long and is always the American Standard Code for Information Interchange (ASCII) characters with values 29 and 31. The control field is 1 byte long, at present it is the ASCII character with value 64, an "@". The control field is reserved for future use. The size field is 8 bytes long, it is a hexadecimal number and indicates the size of the compressed byte array that contains the samples.

Samples in Rados are passive, there is no in memory index or processes that continuously maintain tables, where as the samples in the database are active. It is much faster accessing an active sample than it is to access a passive sample. But keeping samples active requires more resources. Rados gives us acceptable read and write performance, with very low maintenance overhead. In the case of an archive for long-term data storage the trade-off between speed and keeping samples passive was worth it.

Repack and Replication

Everyday the previous day's objects in Rados will be repacked, the repack process reads in the whole object and writes a new object with the same name. The repack process creates a more densely packed object, it takes advantage of the increased cardinality and thus has a much better compression ratio. Compression settings were chosen to have objects with a large compression ratio but foremost to have very good decompression speed, at the cost of higher compression requirements.

After an object has been repacked, the object is marked to be synchronised to the CHPC in Cape Town. A process running on the Katstore node in the CHPC will periodically connect to the Karoo system and copy objects that have been marked ready for synchronisation. The synchronisation process is throttled to only copy a limited amount of objects concurrently. This throttling results in the synchronisation process running for a longer duration but constrains the bandwidth usage over the Wide Area Network (WAN) link connecting the telescope in the Karoo with the CHPC in Cape Town.

QUERY INTERFACE

MeerKAT has a sophisticated Graphical User Interface (GUI) [19] that allows for user friendly control of the telescope and provides advanced analytical tools to telescope operators, scientist, and engineers.

MeerKAT GUI has a dedicated interface for searching and plotting current and historical sensor information, SensorGraph. This interface connect to Katstore's Hypertext Transfer Protocol (HTTP) Representational State Transfer (REST) interface. SensorGraph is the primary interface for users to interact with Katstore.

In addition to SensorGraph, Katstore can also be accessed in several other ways. A Graphite [20] like interface was developed using the graphite-api [21], this allows tools designed to work with graphite to also work with Katstore.

We used the user friendly analytical platform Grafana [22], to create dashboard displays. Grafana connects to the Katstore graphite interface.

For bulk operations we created a very simple Hypertext Markup Language (HTML) interface that allows users to click through to a point where they can download a JSON or Comma-Separated Values (CSV) file containing all the samples of a sensor for one day. This downloaded file maps directly to the sensor sample object stored in Rados, the decompression and unpacking is done server side. This interface implements the best practices for designing a RESTful interface, which enables integration with external applications. Users that are interested in the full resolution of sensor samples over several days or weeks will use this interface.

The query interface was designed to scale, several instances of the query interface can be started on different nodes. Using Ceph Rados as the long term store provide excellent read scalability, objects are read from one of the replicas, thus distributing the load over all the servers that form part of the storage cluster.

SQL

Having the complete sensor history accessible through a database like PostgreSQL not only gave us a stable platform to build on but provided us with full access to the complete SQL language. SQL is a mature and advanced query language.

Without having to write any complicated back end analytical tools we were able to quickly adapt to users needs with regarding to how they wanted the data queried and re-sampled, by extending the SQL queries we used. We do not provide users with direct access to SQL, but the development team adding new functionality to SensorGraph and MeerKAT GUI maintain the SQL queries as part of the application code.

PERFORMANCE

We have not yet taken Katstore through a scientifically rigorous set of benchmarks or done thorough performance analysis on it. But in our current deployment it operates with acceptable throughput.

In our lab setup, the database server is a Dell R430 with 64 GB Random-Access Memory (RAM) and a 256 GB Non-Volatile Memory Express (NVMe) SSD as the database drive. We are able to consistently process 90000 samples a second. The Ceph cluster used in this setup is significantly smaller than the production cluster. The Ceph cluster consisted out of four Dell R420 servers with 48 GB RAM and each server provides three 1 TB HDD to Ceph, each server had only two 1 GbE network interfaces. While maintaining the rate of adding samples to the database we were able to perform 64 simultaneous queries, each to a different sensor, with results returned at a rate of over 8640 samples a second.

CONCLUSION

Although Katstore was specifically designed and developed for the MeerKAT radio telescope the flexible nature of the implementation makes it suitable for many types of time series data.

We do not plan to use Katstore as an event log storage and process platform, MeerKAT uses ELK [23], but it has been tested to ensure that it can handle such a work load. The choices made in building Katstore also makes it suitable for log storage. This gives us confidence that new systems in need of time series datastore can use Katstore.

The use of PostgreSQL not only as a database, but also as a platform to develop on made the Katstore implementation a lot easier and has yielded good performance. PostgreSQL can easily be scaled out and has many tools and good resources to do so.

By keeping the long term storage of samples out of the database and storing them directly in the storage system we were able to build a system that could scale at the rate and ease of a storage cluster.

REFERENCES

[1] M.J. Slabber. 'Overview of the monitoring data archive used on MeerKAT'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 1155–1157.

[2] R. S. Booth et al. 'MeerKAT Key Project Science, Specifications, and Proposals'. In: *ArXiv e-prints* (2009), pp. 1–16. arXiv: 0910.2935. http://arxiv.org/abs/0910.2935

[3] N. Marais. 'MeerKAT Control and Monitoring System Architecture'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 247–250.

[4] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Mar. 2014. DOI: 10.17487/RFC7159. https://www.rfc-editor.org/rfc/rfc7159.txt

[5] F.J. Joubert and M.J. Slabber. 'Distributing Near Real Time Monitoring and Scheduling Data for Integration with other Systems at Scale'. In: *16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain*. JACOW, Geneva, Switzerland. Oct. 2017.

[6] M.J. Slabber and M.T. Ockards. 'Illustrate the Flow of Monitoring Data through the MeerKAT Telescope Control Software'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 849–852.

[7] S. Cross et al. 'Guidelines for Communication with Devices'. In: *SKA SA, July* (2012).

[8] G. Klyne and C. Newman. *Date and Time on the Internet: Timestamps*. RFC 3339 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, July 2002. DOI: 10.17487/RFC3339. https://www.rfc-editor.org/rfc/rfc3339.txt

[9] *PostgreSQL*. Sept. 2017. https://www.postgresql.org

[10] *Python*. Sept. 2017. https://www.python.org

[11] *Timescaledb*. Sept. 2017. http://www.timescale.com

[12] *Ceph*. Sept. 2017. http://ceph.com

[13] *Multicorn*. Sept. 2017. http://multicorn.org

[14] S.A. Weil et al. 'Ceph: A scalable, high-performance distributed file system'. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

[15] N. Marais et al. 'Virtualization and deployment management for the KAT-7/MeerKAT control and monitoring system'. In: *14th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'13), San Francisco, USA*. JACOW, Geneva, Switzerland. Oct. 2013.

[16] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Oct. 2013. DOI: 10.17487/RFC7049. https://www.rfc-editor.org/rfc/rfc7049.txt

[17] *Zstandard*. Sept. 2017. http://facebook.github.io/zstd

[18] *Blosc*. Sept. 2017. http://blosc.org

[19] M. Alberts and F. Joubert. 'The MeerKAT Graphical User Interface Technology Stack'. In: *15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia*. JACOW, Geneva, Switzerland. Oct. 2015, pp. 1134–1137.

[20] *Graphite*. Sept. 2017. https://graphiteapp.org

[21] *Graphite-API*. Sept. 2017. https://github.com/brutasse/graphite-api

[22] *Grafana*. Sept. 2017. https://grafana.com

[23] *Elasticsearch Logstash Kibana*. Sept. 2017. https://www.elastic.co/products