A Processing Pipeline for High Volume Pulsar Candidate Data Streams

R. J. Lyon[a,*], B. W. Stappers[a], L. Levin[a], M. B. Mickaliger[a], A. Scaife[a]

[a]*School of Physics and Astronomy, University of Manchester, Manchester, M13 9PL, UK, Tel.: +44 (0) 161 275 4202*

Abstract

Pulsar candidate analysis pipelines have historically been comprised of bespoke software systems, supporting the off-line study of data. However modern data acquisition systems are making off-line analyses impractical, as they output candidates in such large volumes that they become prohibitively expensive to retain. To maintain processing capabilities when off-line analysis becomes infeasible due to cost, requires a shift to on-line data processing. This paper makes four contributions facilitating this shift relevant to the search for radio pulsars: i) it characterises for the modern era, the key components of a pulsar search candidate processing pipeline, ii) it examines the feasibility of implementing on-line candidate filtering via existing tools, iii) problems preventing an easy transition to on-line filtering are identified and explained, and finally iv) it presents a new prototype filtering pipeline capable of overcoming such problems. Realised using Commercial off-the-shelf (COTS) software components, the deployable system is open source, simple, scalable, and cheap to produce. It has the potential to achieve candidate filtering design requirements for the Square Kilometre Array (SKA), illustrated via testing under simulated SKA loads.

Keywords: pulsars: general, methods: data analysis, methods: statistical, techniques: miscellaneous

1. Introduction

Pulsar search pipelines are comprised of two principal components. The first is a signal processing system. It converts the voltages induced in the receiving element of a radio telescope into digital signals, and identifies those 'significant' detections rising above the noise background. The signal processor (SP) corrects for phenomena such as signal dispersion, excises radio frequency interference (RFI), and optimises search parameters yielding higher signal-to-noise ratio (S/N) detections. The SP ultimately produces some number of pulsar 'candidates' for analysis. These are time and frequency averaged data products describing each detection. For a more complete description of the search process refer to [53].

The second search component is a filtering system. It identifies those candidates most likely arising from legitimate astrophysical phenomena, as opposed to background noise, or terrestrial RFI. In principal this system allows signals of legitimate scientific interest to be isolated and set aside. We refer to this processing component as the data processor (DP). This paper focuses on the development of a new data processor. In particular we contribute the first realisation of a Commercial off-the-shelf (COTS) based DP for pulsar candidate data designed to operate within an incremental data stream (i.e. 'tuple-at-a-time' or 'clickstream' data). This represents an advancement over the current state-of-the-art, and a departure from traditional off-line batch methods.

1.1. Related Work: Pulsar Data Processors

Modern DPs are comprised of custom software tools, written by research groups in the radio astronomy community [e.g. 84, 52, 43, 70]. These evolve over time, accommodating new algorithmic advances as appropriate. It is not uncommon for such advances to result in the discovery of important, previously unknown phenomena. Where existing tools are perceived to fall short, new ones emerge [e.g. 76]. Emerging tools are often tailored to achieve specific science goals (e.g. improve sensitivity to longer period pulsars), and are often written with specific observing setups, or instruments in mind [issues reviewed and described in 58, 59]. There are multiple DP systems in use at any one time, spanning global pulsar search efforts.

Almost all existing DP back-ends execute tasks sequentially on batches of observational data [58, 59]. The batches are usually processed off-line [72], either upon completion of a survey, or at the end of an observational session [e.g. 2, 75, 21, 14]. Recently real-time filtering pipelines have emerged for transient events [63, 78], fast radio bursts (FRBs) [54, 40, 48, 18, 69, 39], pulsars [65], and other phenomena [10, 24, 19]. The adoption of real-time pipelines has occurred due to changing science requirements (desire/necessity for rapid follow-up), and in response to data storage pressures. These pressures are characterised by increasing data capture rates, which in turn yield increasing

*Corresponding author
Email address: robert.lyon@manchester.ac.uk (R. J. Lyon)

volumes of data [55, 56, 59, 72]. Data pressures make it increasingly difficult to process data off-line due to data storage costs [72, 59]. Such challenges are now impacting the search for pulsars. Without the capacity to continually store new data, search methods must be redesigned to maintain their effectiveness as the transition to real-time processing occurs [9, 25, 83, 24].

1.2. Paper Structure

Sections 2 and 3 introduce the core research problem, and describe the standard components of an off-line candidate search pipeline (first contribution). After considering the feasibility of transitioning to on-line candidate filtering (second contribution), and setting out the associated problems (third contribution) Sections 4-5 consider software tools useful for building a new on-line filtering pipeline. Section 6 goes further, and presents a new prototype system (final contribution). Sections 7-9 evaluate the prototype, and present results describing its performance. Finally Section 10 concludes the paper, and reviews the work.

2. Off-line Problem Definition

The goal for a DP is to reduce a set of candidate detections C, to a subset of promising candidates C' expected to possess the most scientific utility. The input set contains N elements, and N varies according to the exact processing configuration used[1]. The DP should return a set C', such that $|C'| \leq |C|$, though we desire $|C'| \ll |C|$.

Each element in C is describable as a candidate tuple. A tuple is a list of m elements. An individual tuple is defined as $c_i = \{c_i^1, \ldots, c_i^m\}$. Here each element is uniquely identifiable in C via the index i. For all $c_i \in C$, it holds that $|c_i| > 0$. For simplicity all $c_i^j \in \mathbb{R}$, and there is no implicit ordering over C. In practice the numerical components of a tuple represent a detection, or its derived characteristics. The standard components usually stored within c_i are shown in Figure 1. The integrated pulse profile shown in a), is an array of continuous variables describing a longitude-resolved version of the signal averaged in time and frequency. The DM curve shows the relationship between candidate S/N and the dispersion measure (DM) [53]. Persistence of the detection throughout the time and frequency domains is shown in c) and d).

Persistence in frequency is represented by a 2-d matrix c), showing pulse profiles integrated in time for a set of averaged frequency channels (i.e. not full frequency resolution). Persistence through time is represented by a 2-d matrix d), showing the pulse profile integrated across similarly averaged frequency channels as a function of time. Other characteristics are also recorded. These can include the S/N, the DM, pulse period, pulse width, beam number, and acceleration; though other metrics are used [58]. The DP uses these information sources to make filtering decisions.

We note that C is not a multi-set, i.e., it does not contain exact duplicates. There are however non-exact duplicates present. These include pulsars detected by different telescope beams, or at slightly different characteristic values. The 'best' detection usually has the highest S/N. When C is available off-line, the 'best' detection is found via exhaustive comparisons in the worst case.

2.1. Existing Data Processors

Existing pipelines execute filtering tasks sequentially upon C. The data is processed either after the completion of a pulsar survey, or at the end of an individual observational session if a faster pace of discovery is desired [see for example, 11, 68, 21, 14]. At present it is feasible to permanently store all candidates in C. This allows detections to be processed more than once. It is common for new pulsars to be found in C, even after it has been searched multiple times [e.g. 41]. This happens when improved search algorithms/parameters are applied, revealing previously hidden detections.

There are some components common to all pulsar DP pipelines. These include 'sifting', known source matching, feature extraction, candidate classification and candidate selection. Together these successively filter candidates. The key components are described in the sections that follow, so their function and computational requirements are understood.

2.1.1. Sifting

Until this work, sifting has only been tackled as an off-line matching problem. The goal is to to accurately identify duplicate detections in a candidate data set (i.e. find harmonically related duplicates), via comparing candidate pairs. For each arbitrary pair c_i and c_k, where $i \neq k$, a naïve sift will exhaustively compare all possible pairs using a similarity measure s. The measure is normally associated with a decision threshold t, applied over one or more variables in a candidate tuple. If s is above some threshold, the pairing is considered a match. Otherwise the pair is considered disjoint.

The accuracy of the similarity measure can be quantified, when the ground truth matching is known. In such cases the performance of s can be measured using a simple metric p, such as accuracy of the output matching,

$$\text{Matching Accuracy} = \frac{\text{Total matched correctly}}{|C'|}. \quad (1)$$

A commonly used sift implementation called 'best' [found in Sigproc, 52] performs an optimised comparison of each

[1] For one observation $N \geq 1000$, for a survey $N > 10^6$ is normal.

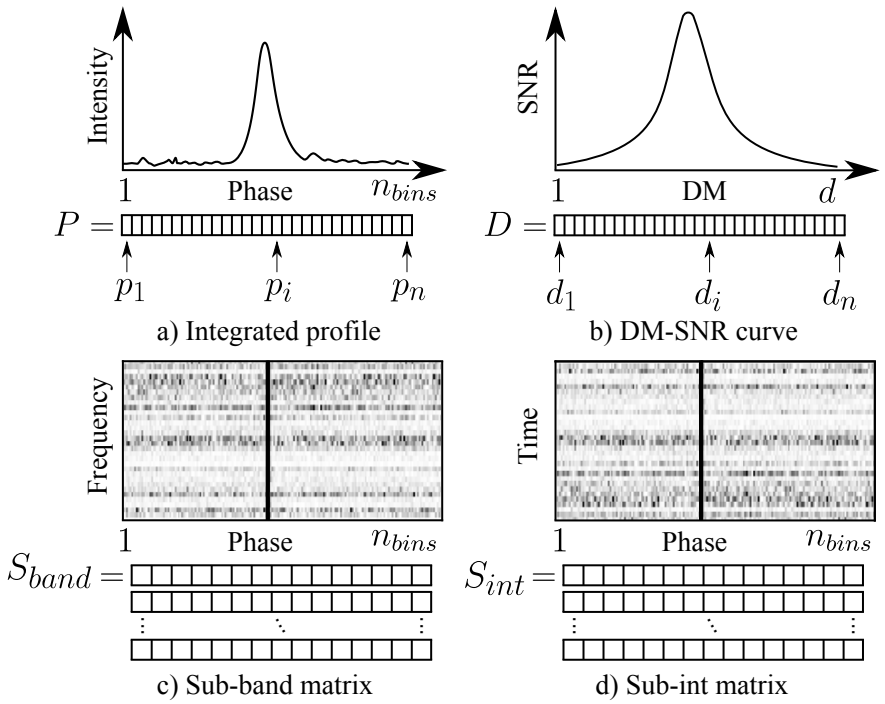

Figure 1: Diagram showing the components of a basic pulsar candidate. Plot a) shows the integrated pulse profile, plotted from the vector P. Plot b) shows the DM-SNR curve obtained from the vector D. Plot c) shows the sub-band matrix, describing the persistence of the signal in frequency. The sub-integration matrix in d) is similar, expect it describes the persistence of the signal in time.

c_i, to every c_k in C. This has a memory complexity of $\mathcal{O}(n)$ as all n candidates must be stored in memory. The corresponding run time is approximately $\mathcal{O}(n^2)$. We note that minor adjustments to the approach can yield better runtime performance. Using combinatorics we find that a decreasing number of comparisons only need be done for each c_i. The total number of permutations for a set of length n, where k items are compared at a time, is given by the binomial coefficient,

$$\frac{n!}{(n-k)! \cdot k!}. \tag{2}$$

This approach is dominated by $\mathcal{O}(n!)$ for all k. In practice for the required $k = 2$ the runtime is dominated by $\mathcal{O}(\frac{1}{2}(n-1)n)$, an improvement over the worst case. This also assumes no domain knowledge is used to avoid unnecessary comparisons.

2.1.2. Feature Extraction

Numerical variables known as 'features' are usually extracted from candidates post-sifting. These are useful for deriving accurate filtering decisions, and are stored within each candidate tuple. Historically, features were comprised of standard signal characteristics (DM, pulse width, etc) used by human experts to filter candidates manually. Today features are utilised principally by ML algorithms [see 30, 15, 71]. These build mathematical models able to filter and separate candidate data automatically, based on their feature values. For pipelines employing ML tools, features are generally more complicated than simple signal characteristics (see Section 2.1.3 for more details). The computational cost of extracting features varies. The costs are rarely reported, though can be very high [58].

2.1.3. Candidate Selection

Candidate selection applies filtering decisions using the features that comprise a candidate tuple. Candidates passing through this stage are 'selected' for further processing or study. Candidate selection varies in complexity. From a single threshold applied over S/Ns, to far more complex automated machine learning-based approaches [e.g 31, 12, 90, 64, 59]. Where only thresholds are applied, runtime and memory complexity is constant per candidate. For machine learning and other more sophisticated methods, complexities are hard to determine as they are input data dependent. Candidate selection is notoriously difficult. In recent years a rise in candidate volumes has spawned what has become known as the 'candidate selection problem' [31].

2.1.4. Known Source Matching

Known source matching involves determining which detections correspond to known pulsar sources. The procedure is often carried out during sifting, though can be done independently after candidate selection. It requires a set K of known sources, usually obtained from a known pulsar

catalogue [such as, 62]. For simplicity[2] each source in K is defined as $k_i = \{k_i^1, \ldots, k_i^m\}$, with each tuple uniquely identifiable in K via the index i. As before for candidates, all $k_i^j \in \mathbb{R}$, with the meaning of each k_i^j the same as for candidates (or at least mappable). A brute force matching approach compares each candidate c_i to every $k_i \in K$. This corresponds to a runtime complexity of $\mathcal{O}(n \cdot |K|)$. As new pulsars (and other possible radio sources) continue to be found over time, $|K|$ is gradually increasing. The brute force method is thus computationally expensive[3] for an increasing $|K|$ and an unbounded n. The memory complexity is $\mathcal{O}(|K|)$, as each known source is typically stored in memory to facilitate fast matching.

2.1.5. Manual Analysis

Following the previous steps, a set of candidates C' will be stored for manual analysis. Here experts manually examine the elements of C', and judge their discovery potential. Given the large number of candidates usually in C', this process is time consuming. This in turn introduces the possibility for human error [59]. Manual processing steps may have to be re-run when optimised search parameters/improved search methods are applied to C'.

2.2. Feasibility of Real-time Search

Empirical data describe a trend for increasing survey data capture rates over time [59]. This is expected to continue in to the Square Kilometre Array (SKA) era [74, 28, 16, 58]. The projected data rates are large enough to make the storage of all raw observational data impossible (e.g. filterbank/voltage data), and the storage of all reduced data products (e.g. time vs. frequency vs. phase data cubes) impractical [59]. Without the capacity to store data for off-line analysis, SKA pulsar searches will have to be done in real-time at SKA-scales [28, 72, 58].

To overcome this problem, selection methods must be adapted to the real-time paradigm. A real-time DP must operate within the time and resource constraints imposed by instruments such as the SKA [26, 27, 67, 77]. DP operations must also execute functionally within those constraints. This is difficult to achieve. Not all DP operations, as currently deployed, can run within a real-time environment. The following problems reduce the feasibility of transitioning off-line pipelines to the on-line paradigm:

- Sifting is an operation that requires data to be aggregated. This is problematic in a real-time environment. The time taken to aggregate data can violate latency requirements. Whilst the memory required to retain all N candidates may not be available. For observations where rapid follow-up is crucial (fast transients), the delay induced via aggregation delays a rapid response.
- Pulsar feature extraction has never been undertaken on-line, and is currently an off-line batch process.
- Known source matching, similar to sifting, is currently executed over all candidates off-line. It therefore suffers from similar issues.
- The most accurate selection systems in use today, are built using off-line machine learning algorithms. Such off-line systems are inherently insensitive to distributional changes in the input data over time. Yet sensitivity to change is an important characteristic for an on-line system to possess, as it enables automated learning and operation.

Given such challenges, existing methods must be redesigned, or new techniques developed, to maintain the effectiveness of DPs for future pulsar searches. In some cases converting pipeline tasks to run on-line is deceptively trivial. For instance sifting could be done on-line, via only comparing each c_i to c_{i-1} and c_{i+1}. Whilst functionally possible, such an approach would significantly reduce sifting accuracy. This would in turn produce many more false positive detections, increase the lead-time to discovery, and possibly preclude some discoveries being made.

3. On-line Problem Definition

In a data streaming environment, only a portion of C is available for processing at any given time. Either due to real-time constraints forcing batches of data to be processed separately, or C being too large to process in a single pass. In either case there are two processing models that can be employed [discussed elsewhere e.g., 91, 17]. The incremental 'tuple-at-a-time' model applies when individual data items arrive at discrete time steps. An item c_i arriving at time i, is always processed after an item arriving at time $i-1$, and before $i+1$. The batch (or micro-batch/discretized streams) model applies when groups of items arrive together at discrete time steps. Here batch B_i arrives after B_{i-1} and before B_{i+1}. This is summarised in Figure 2 for clarity. Both models temporally order data, which has implications for how the data can be processed.

The batch model is advantageous when groups of items exhibit distributional similarity. For instance when processing data from seismic sensors, it makes sense to work with batches, as patterns of seismic activity will likely be close temporally. Whilst if looking for fraudulent activity in a stream of random financial transactions, each transaction should be processed in isolation. Otherwise an innocent transaction may be incorrectly linked to a fraudulent one. There is a trade-off between computational efficiency and practical utility [20, 51], that must be struck when choosing a processing model.

For the processing of single pulse events, it does not make sense to batch data. Waiting for a batch incurs a

[2] Known sources are well studied, thus more information is available describing them, than for candidates.

[3] It does not reach quadratic complexity, as $|K| \ll n$.

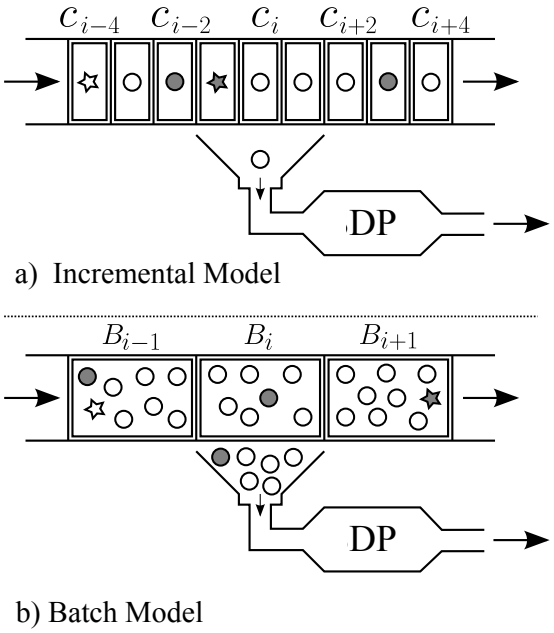

Figure 2: Incremental and batch data processing models. The shapes moving through the models represent different item categories (e.g. pulsar vs. non-pulsar). The true class labels are known a priori for the shaded items only.

time penalty which impedes our ability to initiate rapid follow-up. For pulsar search batch processing is possible, however batch sizes are potentially very large. In both cases it would appear that an incremental model is the simplest to adopt.

3.1. An Incremental Model for Data Processing

We extend the notation used previously. The candidate set C is now a candidate stream $C = \{c_1, c_2, \ldots, c_n, \ldots\}$. Here each c_i is delivered at time step i, and c_i always arrives before c_{i+1} (discrete time model with temporal ordering). The size of the set C is unbounded in this scenario (unknown N). This assumes a limitless supply of individual candidates, redefining the SP as a data stream producer, and the DP as a data stream consumer.

3.2. Practical Assumptions

We assume a worst case scenario, where each data item is processed just once, as per the incremental model. Filtering decisions must be made on an individual candidate basis, with limited knowledge. For example, suppose c_2 arrives for processing where only c_1 has been observed. Here a filtering decision can only be made using knowledge of the candidate/s already seen and c_2. If c_2 is most likely noise or interference, then the decision is conceptually simple. Yet the decision making process becomes complicated in scenarios where multiple detections of the same source are likely to be made during an observation. If c_2 is indeed a pulsar detection, do we retain it? Is it the strongest detection we are likely to see? If it is weak, should we throw it away, assuming a stronger detection will come along? Do we keep every weak detection, and risk increasing the size of C'? There are no simple answers to such questions.

To proceed in our domain, we assume candidates will be ordered according to c_i^0 (an ordering variable). This ordering must be strict, so that $\forall c_i \in C$, $c_i^0 \leq c_{i+1}^0$. This definition specifies an ascending order, though it would make no difference if it were descending. By using an ordering we can have confidence that similar items will be close together in the stream (close temporally). This simplifies our data processing, and will be used to develop an on-line sifting algorithm in Section 6.2. Note that we are implicitly assuming there exists an upstream processing component, capable of ordering data items correctly (ordering applied across 1 or more input streams). We accept that it may not always be possible to assign an ordering conducive to improved data processing. However, where possible we should attempt to order our data in a meaningful way, as it greatly assists with the downstream processing. We now look for frameworks that can accommodate this processing model.

4. Candidate Frameworks for Prototyping

The design of a DP is driven by multiple considerations. We have focused upon utilising COTS software components to reduce cost. This is our primary design driver. However our choice of software components was also driven by the need to reduce computational overheads, whilst maximising scalability and design modularity.

4.1. Review & Chosen Framework

Signal processing systems often utilise accelerator hardware to enable real-time operation. This includes Graphics Processing Units (GPUs) [e.g. 61, 23, 28] and Field Programmable Gate Arrays (FPGAs) [e.g. 44, 29, 89, 85, 86]. In contrast most DP software is executed on general purpose computing resources, with some exceptions[4]. We therefore only consider DP execution upon standard Central Processing Units (CPUs). There are many frameworks/approaches that meet our criteria [for a review see e.g., 32]. Yet we chose Apache Storm [35, 4] to support our new on-line prototype. It was chosen due to its application in the real-world to similar large-scale incremental processing problems [see 37, 81, 87] and strong user support base. Storm is similar to frameworks such as Apache Hadoop [88, 36, 7], Samza [45, 6], Spark [92, 8] and S4 [66, 5], and is fairly representative of similar modern COTS tools. Since completing this work, we discovered other frameworks in the software ecosystem similar to Storm. These include Apache Samoa [13, 3], Apache Flink [17], and Kafka Streams [73]. The use of Apache Storm in lieu

[4]Machine learning based filters [e.g. 90].

of these systems, does not undermine the novelty or usefulness of the contributions of this work. Principally as the processing pipeline proposed in Section 6., is framework independent and compatible with any tool capable of supporting the 'tuple-at-a-time' processing model.

5. Apache Storm-based Processing Framework

Storm is a Java-based framework. It operates under an incremental data stream model. It is underpinned by the notion of a directed-acyclic graph [DAG, see 80]. The graph models both the processing steps to be completed, and the data flow. The graph is known within Storm as a *topology*. Storm topologies only allow data to flow in one direction. This makes recursive/reciprocal processing steps impractical to implement (though newer frameworks discussed in Section 4 can overcome such limitations).

5.1. Topologies

A topology is comprised of nodes. Nodes represent either data output sources known as *spouts*, or processing tasks applied to the data called *bolts*. Data flows from spouts to bolts via edges in the topological graph. Individual data items are transmitted in the form of n-tuples. These are finite ordered lists much like the candidate tuples defined in Section 2.

5.2. Edges

Edges represent generic connections between processing units, made via a local area network (LAN), or a wide area network (WAN). Data flows via the edges as tuples. Upon arriving at a bolt a tuple is usually modified, and the updated tuple emitted for a downstream bolt to process. The practical data rates between edges are non-uniform, and often subject to change. Fluctuating data rates can lead to resource contention if too much data is funnelled through too few processing bolts.

5.3. Bolts

Each bolt encapsulates a modular computing task, completable without interaction with other bolts. Such modularity allows data to be processed in parallel, across heterogeneous computing resources without a loss in functionality. The duplication of bolts allows the topology to be scaled to increasing amounts of data without additional development effort. It also means that when a bolt fails, it can quickly be replaced with a new instance. The modular design is not ideal for all scenarios where persistence of state is required. Generally state persists within a spout/bolt only whilst it is executing. If it fails or is stopped, that state is lost without external state management. For processing components that, i) are not lightweight, ii) require a persisted state to function correctly, or iii) require data to be aggregated/batched, Storm can be more difficult to apply.

5.4. Tuple Flow

It is possible to control the flow of tuples through a topology. This is achieved via a cardinality relation. This is defined between spouts and bolts via edges. The relation can be one-to-one, one-to-many, or one-to-all. If using a one-to-one relation, tuples either move forward to a single random bolt, or a value-specific bolt based on a tuple attribute test[5]. Upon transmission of a tuple, a 'tuple received' acknowledgement can be requested by the sender (originating spout or bolt). If the sender receives no acknowledgement, it will resend the tuple after some time-out period has elapsed. This enables Storm to maintain the property that all data is processed *at least* once even in the event of failure [see 35], helping ensure fault tolerance.

5.5. Deployment

Topologies execute upon clusters comprised of two distinct nodes. There are i) master nodes that run a daemon called 'Nimbus', and ii) one or more worker nodes running a daemon called 'Supervisor'. A master node is responsible for executing tasks on the worker nodes, and restarting spouts/bolts after failures. The worker nodes execute spouts and bolts separately, within threads spawned by a Java Virtual Machine (JVM).

Communication between the master and worker nodes is coordinated by Apache Zookeeper [36]. Zookeeper maintains any state required by the master and the supervisors, and restores that state in the event of failure. A Storm cluster therefore consists of three components: Nimbus, one or more Supervisors, and a Zookeeper.

6. DP Prototype Design

The prototype design is shown in Figure 3. It has a single 'layer' of input spouts, which emit candidate tuples to random downstream bolts (arriving at the data ingest layer) at a controllable rate. Candidates then propagate to each subsequent layer in the topology. Either at random to ensure tuples are load balanced[6] across the bolts, or according to some custom stream partitioning as in Section 6.2. The first layer contains data ingest bolts. These modify the data so that it is transmitted in a format amenable to further processing.

The second layer attempts to filter out duplicate candidates via sifting. To achieve this, the sifting bolts employ a new distributed global sift algorithm. This is described in more detail in Section 6.2. After sifting, tuples are sent to bolts that perform data pre-processing (i.e. normalisation). This is required prior to ML feature extraction.

[5] For example if $c_i^1 \leq 10$, send to bolt b_1^1, else to bolt b_1^2.
[6] See [35] for more details of load balancing in Storm.

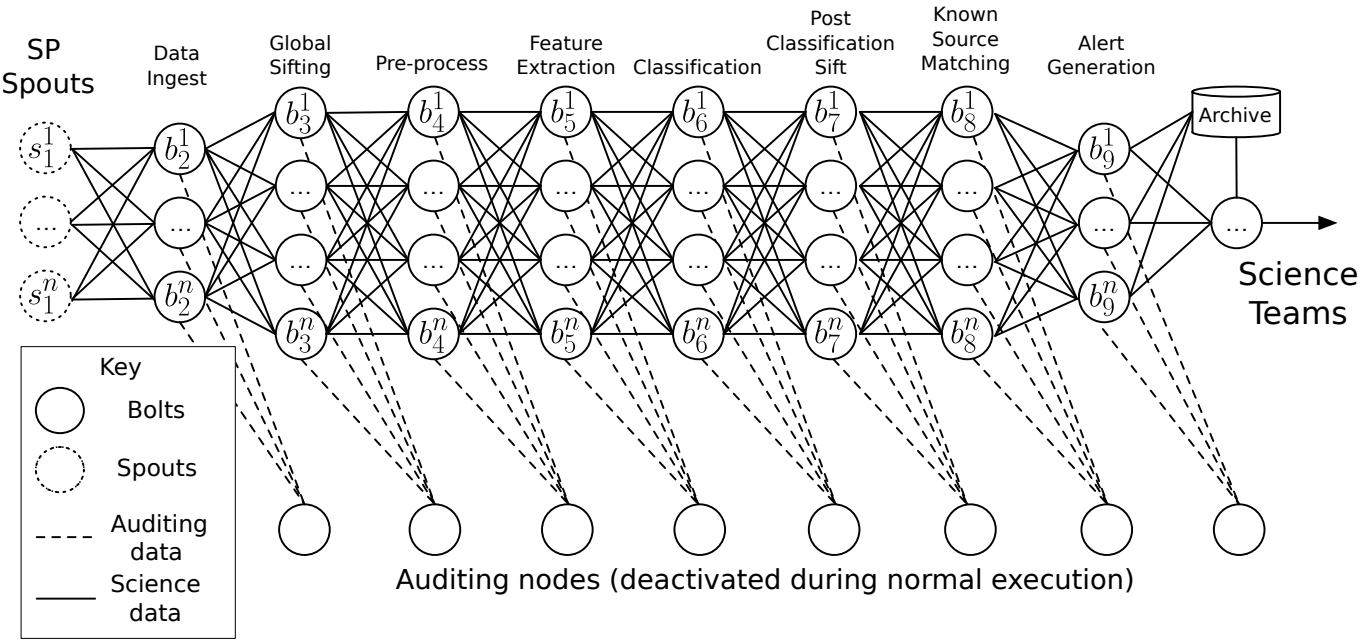

Figure 3: The prototype DP topology. Data is supplied to the topology via SP spouts, which generate candidate tuples. The tuples are propagated forwards through the topology. No tuple is duplicated, and each reaches only one bolt in the topology at any given time. There are nine distinct types of bolt in the topology, described in the sections that follow.

Once complete, the next layer of bolts extracts the features. The features are appended to the tuple, and sent onwards for further processing.

Following feature extraction, tuples are passed to ML classification bolts. These execute an on-line ML algorithm developed by [59]. This predicts the true class origin of each candidate. The predictions are appended to each tuple, and passed on to secondary sifting bolts. These remove duplicates in lieu of the additional information obtained during ML classification. Tuples making it through the second sift, are passed to source matching bolts. These attempt to match promising candidates to known pulsar sources in a pulsar catalogue. Candidates not matched to known sources (likely new pulsars) generate alerts for follow-up action.

The topology has been designed to provide auditing capabilities. Auditing nodes are connected to the processing bolts, via the dashed lines shown in Figure 3. They are used to audit the performance of individual bolts, or the entire topology during testing. The auditing nodes are not activated during normal execution.

The individual spouts/bolts are now described in more detail. Source code for the procedures described in the following sections can be found on-line [60]. The code is provided in the form of an interactive iPython notebook. It contains additional details that support the information presented in this paper.

6.1. SP Spouts

At present no SP system can generate data fast enough to stress our prototype. We therefore created SP spouts which emit real candidate tuples driving the data processing. These play the role of the SP, delivering data at a controllable rate. The spouts are highly customizable. They can generate a fixed number of candidates, or as many as possible within a fixed period of time. Importantly the ground truth label for each and every generated tuple is retained. This allows candidate filtering accuracy to be evaluated anywhere within the topology.

To be realistic, the spouts must output real-world candidate class distributions. For the prototype we define a ratio used to achieve this. Suppose each candidate c_i is associated with a label defining its true origin. This can be modelled numerically via a binary variable y, where $y_i \in Y = \{-1, 1\}$. Here $y_i = -1$ equates to non-pulsar and $y_i = 1$ to pulsar. The set of candidates which describe real pulsar detections $P \subset C$ contains only those candidates for which $y_i = 1$. Similarly $\neg P \subset C$ contains only those candidates for which $y_i = -1$. Thus the ratio,

$$c_{\text{ratio}} \quad = \quad \frac{|P|}{|\neg P|}, \tag{3}$$

describes the imbalance between pulsar and non-pulsar candidates in generated data. The ratio observed during real pulsar searches varies from 1:7,500 [42, 79] to 1:33,000 [22, 49, 1]. For the prototype, we maintain ratios of up to 1:10,000 (0.0001). Evidence suggests this figure to be representative [59] and challenging to deal with.

Variable	Value
t_{obs}	600 seconds
n_{beam}	1,500 (SKA-Mid)
c_{beam}	1,000
c_{obs}	1,500,000
c_{size}	2.2 MB
c_{ratio}	0.0001
d_{rate}	5.5 GB/s
c_{rate}	2,500 (per second)
d_{volume}	3.3 TB

Table 1: Summary of assumptions made when designing the prototype. Here t_{obs} is the observation time in seconds, n_{beam} the number of beams used per observation, c_{beam} the total number of candidates anticipated per beam, c_{obs} the total number of candidates anticipated per observation, c_{size} the size of an individual candidate, c_{ratio} the ratio of pulsar to non-pulsar candidates in input data, d_{rate} the anticipated data rate per second, and finally d_{volume} is the expected total data volume per observation.

To maintain the ratio, spouts have been designed to emit two types of candidate. Type 1 candidates are contrived randomly generated non-pulsar tuples. Whilst valid, they simply resemble white noise. Type 2 candidates are representations of real candidates. These were sampled from data obtained during the High Time Resolution Universe Survey South [HTRU, 42, 57]. Type 2 candidates provide a genuine test of our prototype's discriminative capabilities. Type 2 candidates can describe either the pulsar or non-pulsar class. As the sample of type 2 candidates is small, some pass through the pipeline more than once during testing. This does not invalidate our results, as duplicates must be passed through the system to test our sifting approach.

6.1.1. Spout Data Rate and Volume

Input data rates are determined by the length of an observation t_{obs} (seconds), the number of telescope beams n_{beam}, the number of candidates returned by the SP per beam c_{beam}, and the size of each candidate c_{size} (MB). The total data volume per observation is given by,

$$d_{\text{volume}} = n_{\text{beam}} \times c_{\text{beam}} \times c_{\text{size}}. \quad (4)$$

Whilst the SP to DP data rate, assuming a steady uniform transmission, is given by,

$$d_{\text{rate}} = \frac{d_{\text{volume}}}{t_{\text{obs}}}. \quad (5)$$

The assumed parameter values are given in Table 1. These were chosen after studying SKA design documentation [see 26, 27, 16]. Here candidate tuples are comprised of,

- a 262,144 sample data cube (128 bins, 64 channels, 32 sub-ints), where each sample is 8 bytes in size. This describes the detection in time, phase, and frequency.
- the variables; right ascension (RA), declination (DEC), S/N, period, DM, beam number. These variables require 56 bytes of storage space.

Together these components produce a tuple ≈ 2.2 MB in size. For 1,500 beams, and 1,000 candidates per beam, this equates to 1.5 million candidates. The corresponding total data volume is ≈ 3.3 TB per observation. The $t_{\text{obs}} = 600$ second time constraint, implies a data rate of 5.5 GB/s. The SP spouts will generate data at and above this rate, when simulating pulsar search operations.

6.2. On-line Global Sifting

The simplest sift approach only compares candidates detected within the same telescope beam [e.g. 38]. Our approach works globally across beams, in principle yielding better results. It sifts one candidate at a time, as opposed to candidate batches. It's success is predicated on two assumptions. First, it assumes a stream partitioning that ensures candidates with similar periods from all beams (measured in μs) arrive at the same bolts. Thus an individual bolt represents a period range, and together they cover the plausible range of pulse periods. Second, it assumes candidates are ordered (see Section 3.2) according to their pulse periods. This facilitates the partitioning. Since similar period candidates always arrive at the same bolts, it becomes possible to check for duplicates via counting observed periods. The logic underpinning this approach is simple. If a period is observed many times, it is a possible duplicate. When possible duplicates are compared via other variables (e.g. DM) for additional rigour, counting can be used to efficiently find duplicates.

The general approach is summarised in Algorithm 1. This describes the code at a single bolt. Note the algorithm requires sufficient bins to count accurately. As pulsar periods are known to be as low as 1.396 milliseconds [33], and given that pulse periods can be similar at the microsecond level; there must be enough bins to count at microsecond resolution. The algorithm uses the array F to maintain the count, by mapping the array indexes $0, \ldots, n$ to specific periods/period ranges. The state of F only persists for a single observation. We use a scaling and a rounding operation to achieve this. Together these find the correct bin index to increment for an arbitrary period. The variables *floor* and *ceiling* help to accomplish this. The scaling of the data is done on line 8, with the integer rounding operation done on line 9 which obtains the index.

The if-else statement on lines 10-15 contains the only logic in the algorithm. It checks if the period counter is greater than zero (indicating it has already been observed). If $F[index] > 0$, then we pass this information to the similarity function for checking. The algorithm can be modified for improved accuracy. For example, if frequency counts are maintained for multiple variables (DM, pulse

Algorithm 1 Global Sift

Require: An input stream $C = \{..., (c_i), ...\}$, such that each c_i is a candidate, and c_i^j its j-th feature. Here c_i^0 is the pulse period (ordering variable). Requires a similarity function s which can be user defined, the number of period bins p_b to use, the smallest period value expected at the bolt min, and the largest period value expected at the bolt max.

1: **procedure** GLOBAL SIFT(C, s, p_b, min, max)
2: **if** F = null **then**
3: $F \leftarrow array[p_b]$ ▷ Init. counting array
4: $n \leftarrow 0$ ▷ Init. candidate count
5: $floor \leftarrow 0.0$
6: $ceil \leftarrow p_b$
7: $n \leftarrow n + 1$ ▷ increment observed count
8: $p \leftarrow ((ceil - floor) * (c_i^0 - min)/(max - min)) + floor$
9: $index \leftarrow (int)p$ ▷ Cast to int value
10: **if** $F[index] > 0$ **then**
11: F[index]++;
12: **return** s(true); ▷ Similarity check, period seen
13: **else**
14: F[index]++;
15: **return** s(false); ▷ Similarity check, period not seen

width, beam etc.), the probability of a match can be estimated across all of them. Note the similarity function s used on lines 12 and 15, can also be made to check matches counted in neighbouring bins. Candidates counted in bins neighbouring $F[index]$ have very similar periods, and thus should be considered possible duplicates. It is trivial to implement, though requires additional memory and a sliding window of examples. The use of a sliding window is discussed briefly below.

The general approach described has weaknesses. The first candidates arriving at a sift bolt will never be flagged as duplicates, due to no prior periods being been observed. Furthermore, the approach will treat duplicates with different S/Ns as equivalent. However we wish to retain only the highest S/N detection, and discard the rest. The model described thus far cannot achieve this. It cannot anticipate if, or when, a higher S/N version of a candidate will enter the topology. Such weaknesses are a result of the trade-off between computational efficiency, and sifting accuracy. This problem can however be overcome, using a sliding 'similarity' window over the data [see 60].

6.3. On-line Feature Extraction

Eight features are extracted from each tuple moving through the topology. These are described in Table 4. of [59]. The first four are statistics obtained from the integrated pulse profile (folded profile) shown in plot a) of Figure 1. The remaining four similarly obtained from the DM-SNR curve shown in plot b) of Figure 1.

The computational cost of generating the features in Floating-point operations (FLOP) is extremely low. If n represents the number of bins in the integrated profile and DM curve, the cost is as follows:

- **Mean cost:** 2(n-1) + 4 FLOP.
- **Standard deviation cost:** 3(n-1) + 9 FLOP.
- **Skew cost:** 3(n-1) + 7 floating point FLOP.
- **Kurtosis cost:** 3(n-1) + 7 floating point FLOP.

Assuming that addition, subtraction and multiplication all require 1 FLOP, whilst division and square root calculations require 4. The Standard deviation calculation assumes the mean has already been calculated first. Likewise, the skew and kurtosis calculations assume the mean and standard deviations have already been computed, and are simply reused. The features cost 2,848 FLOP per candidate [58]. The runtime complexity of the feature generation code is $\mathcal{O}(n)$ over the input space, and memory complexity is $\mathcal{O}(n)$. Note that in both cases n is small.

6.4. On-line Classification

This prototype is focused on processing incremental data streams in real-time. At present only one classifier has been developed specifically for incremental pulsar data streams - the GH-VFDT [56, 59, 58]. All other pulsar classifiers developed in recent years can be characterised as off-line supervised systems. Thus in terms of classifiers, the GH-VFDT is the logical choice for this work. Despite this we acknowledge that we may have overlooked classifiers capable of achieving better recall on our data. As the goal of this paper is to illustrate the feasibility of processing pulsar data incrementally in real-time, and not to advocate the use of a specific ML algorithm, this is not something we consider to be problematic. Our design does not preclude the use of any other incremental data stream classifier developed now or in the future.

The GH-VFDT classifier was designed for pulsar candidate selection over SKA-scale data streams. It is an on-line algorithm, capable of learning incrementally over time. It is therefore able to adapt to changing data distributions, and incorporate new information as it becomes available. The GH-VFDT is extremely runtime efficient, as it was designed to utilise minimal computational resources.

The algorithm employs tree learning to classify candidates, described via their features [for details of ML classification see, 30, 15, 71]. Given some ground truth 'training data' describing the pulsar and non-pulsar classes, tree learning partitions the data using feature split-point tests [see Figure 8 in 59]. Split points are chosen that maximise the separation between the classes. In practice this involves firstly choosing a variable that acts as the best class separator. Once such a separator is found, a numerical threshold 'test-point' is found, that yields the greatest class separability. This process repeats recursively, producing a tree-like structure. The branches of the tree form

decision paths. The paths can yield highly accurate classification decisions when given high quality training data.

The memory complexity of the algorithm is $\mathcal{O}(lf \cdot 2c)$ (sub-linear in n), where l describes the number of nodes in the tree, f the number of candidate features used ($f = 8$ for the prototype), and finally c the number of classes (here $c = 2$, pulsar & non-pulsar). The runtime complexity of the algorithm is difficult to quantify, as it is input data dependent. However it is of the order $\mathcal{O}(n)$.

6.5. On-line Post Classification Sift

This aims to remove duplicate detections in lieu of ML predicted class labels. Here two candidates with similar period and DM values (or some other variables), have their predicted class labels compared. If the same, these are considered likely duplicates. In this case, only the 'best' detection need be forwarded on. This approach can be improved using a sliding window over tuples [see 60]. This allows similar tuples to be compared in micro-batches defined by the window. Only after the window moves away from an unmatched tuple does it get forwarded.

6.6. On-line Known Source Matching

We have developed a new, very fast, source matching algorithm possessing lower computational complexity than common off-line matching approaches (see Section 2.1.4). Using tree-search it recursively divides the matching search space, greatly reducing the number of comparisons to be undertaken. It relies on an ordering applied over the set of known sources K (an array), to find a position in K to begin searching that reduces the search space. A total ordering of elements in K is required, according to some variable k_i^j. To achieve a total ordering, then for all k_i^j, k_{i+1}^j, and k_m^j, where $m > i + 1$,

$$\text{if } k_i^j \leq k_{i+1}^j \text{ and } k_{i+1}^j \leq k_i^j \text{ then } k_i^j = k_{i+1}^j, \tag{6}$$

$$\text{if } k_i^j \leq k_{i+1}^j \text{ and } k_{i+1}^j \leq k_m^j \text{ then } k_i^j \leq k_m^j, \tag{7}$$

$$k_i^j = k_i^j, \tag{8}$$

$$k_i^j \leq k_{i+1}^j. \tag{9}$$

Here equations 6-9 define the antisymmetry (6), transitivity (7), reflexive (8) and totality properties (9). To apply the ordering, we require a numerical value per source satisfying these properties. This can be obtained via measuring the angular separation θ, between each known source, and a single reference coordinate ($00^h\ 00^m\ 00^s$ and $00^\circ\ 00'\ 00''$). This allows sources to be strictly ordered according to their separation from the reference point. This reference value should be computed off-line so that the array K is ordered correctly in advance.

For each candidate source c_i to be matched, the reference separation is computed. Known sources near to c_i are similarly separated from the reference point, as too are the known sources antipodal to the candidate. This is depicted in Figure 4. The array index of the known source in K possessing a reference separation closest to that of C_i, is found via tree-search [46]. This array index becomes the 'search index'. From there, we need only compare c_i to sources immediately about the search index, i.e. with separations $\leq 2\theta$ with respect to the reference point. Precisely how many comparisons are done to the left and right of the search index, is up to the user and their desired matching precision. For the prototype, comparisons are not undertaken if the distance between c_i and k_i exceeds 1.5°.

Algorithm 2 Matching Procedure

Require: A known source k_i, a candidate c_i, an angular separation used for matching θ, and an accuracy level for period and DM matching $e_{margin} \in [0, 1]$.

```
1: procedure isMatch(k_i, c_i, θ, e_margin)
2:     c_p ← c_i                          ▷ Get period from candidate
3:     c_dm ← c_i                         ▷ Get DM from candidate
4:     k_p ← c_i                          ▷ Get period from known source
5:     k_dm ← c_i                         ▷ Get DM from known source
6:     p_diff ← (e_margin × c_p)/2
7:     dm_diff ← (e_margin × c_dm)/2
8:     hms ← [1, 0.5, ..., 0.03125]       ▷ Harmonics to check
9:     for h ← 0, h++, while h < |hms| do
10:        if c_p > (k_p * hms[h]) − p_diff then
11:            if c_p < (k_p * hms[h]) + p_diff then
12:                if c_dm < k_dm + dm_diff then
13:                    if c_dm > k_dm − dm_diff then
14:                        sep ← calcSep(k_i, ci)
15:                        if sep < θ then
16:                            possibleMatch(k_i, ci)
```

The matching procedure shown in Algorithm 2, compares the period and DM of a promising c_i, to some potential match k_i. The known source k_i is considered a possible match, only if their period and DM values are similar to within a user specified error margin $e_{margin} \in [0, 1]$. For example, an $e_{margin} = 0.1$ corresponds to a 10% error margin. When using this margin we consider a known source to be possible match, only if its period and DM are within 10% of the candidate's ($\pm 5\%$).

The computational complexity of the approach is $\mathcal{O}(n \cdot \tau)$, where τ is a proxy for the number of comparisons made between known sources and candidates based on θ. The modified runtime is practically speaking linear in n (as τ is usually small). The complete algorithm has been implemented in an iPython notebook [60].

6.7. On-line Alert Generation

Prototype alert generation bolts have limited functionality. These do not generate genuine alerts, since alerts are not required for our prototyping efforts. Thus alert nodes

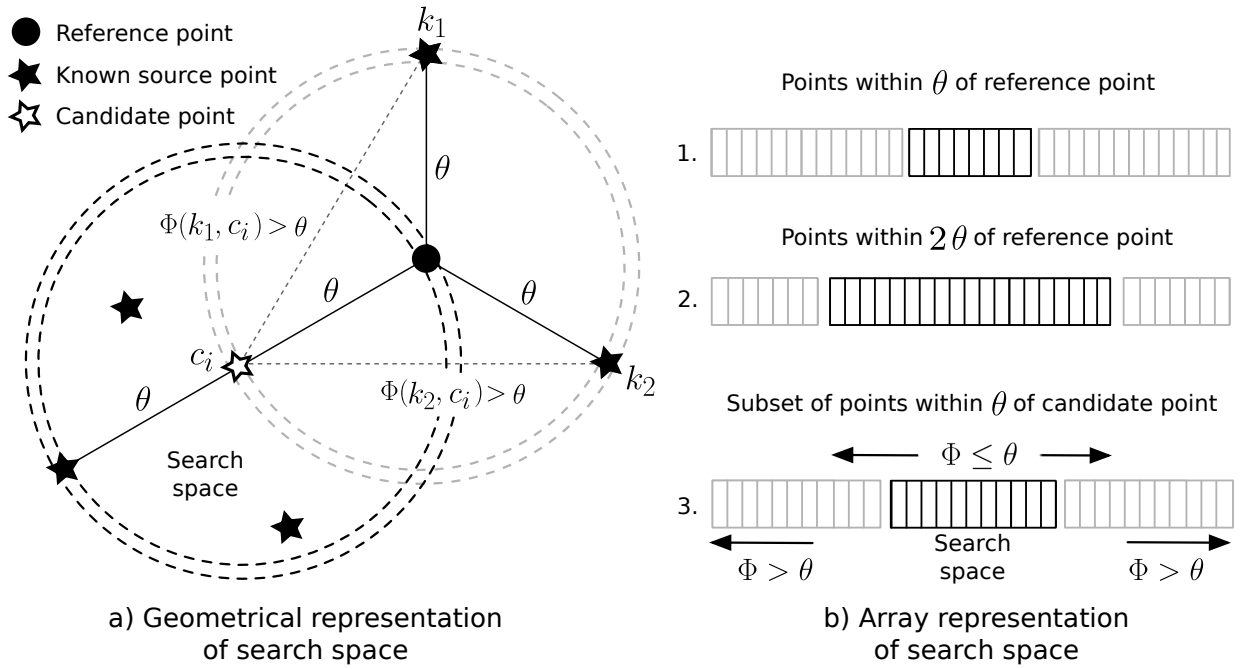

Figure 4: A visual representation of how the matching search space is determined. In a) we see candidate source c_i, which has an angular separation from the reference point of θ. There are also 2 known sources k_1 and k_2, with the same separation from the reference point. The angular separation between c_i and other known sources is computed by Φ. If the separation is greater than θ, then c_i is not compared to k_i. Otherwise a comparison is performed. From this representation it is clear that c_i could be related to known sources up to 2θ from the reference point. However only the region around c_i should be searched. In b) we see the same information with respect to the known source array. The sources in the array are sorted according to their angular separation to the reference point. The ordering allows the search region to be found quickly in step 1. This is then refined, producing the narrowed search region in step 3.

simply act as processing units, that slow down computation as though alerts were being generated. Similarly as no archival system exists, archival is not simulated.

6.8. Auditing

The prototype incorporates processing nodes able to audit runtime/filtering performance. Auditing is accomplished in two ways. Per-node auditing records the filtering accuracy and runtime performance, for a specific node only. This incurs a small computational overhead impacting the node being audited. As this type of auditing is only intended for use during development, it does not affect runtime performance during scale testing.

End-to-end auditing measures filtering accuracy and runtime performance across the pipeline. This is achieved without incurring additional runtime overheads via the use of timestamps. Unique timestamps are attached to each tuple upon entering and exiting the topology. The timestamps accompany tuples through the topology, and record time to microsecond resolution. By determining the difference between timestamps, an estimation of the time taken for an individual tuple to move through the topology can be determined. Individual tuple transfer times can also be aggregated. By averaging over all tuples reaching the end of the topology, average tuple processing times can be computed. Additional metrics are also monitored at the auditing bolts.

7. Simulations

Two forms of simulation were undertaken to test the prototype. The first consisted of small-scale simulations executed on a single machine. These were useful for testing and debugging the topology design and processing code. The second involved a larger scale deployment of the topology to the Amazon Elastic Compute Cloud (EC2). The cloud simulations were intended to assess the scalability of the system, and determine ease of deployment. In both scenarios the goal was to recreate the delivery of data from the SP to DP, during a plausible pulsar search scenario. For both local and cloud simulations, each experiment was executed 10 times and the results averaged, to mitigate the impacts of result variability caused by the hardware used.

7.1. Local 'Cluster' Mode

Simulations were undertaken on single a computer running OSX 10.9. It possessed a single 2.2 GHz Quad Core mobile Intel Core i7-2720QM Processor, with a peak theoretical performance of 70.4 GFLOPs [34]. It was equipped with 16 GB of DDR3 RAM, and two 512GB Crucial MX100 solid state drives. A Storm cluster (version 0.95) was deployed on this machine, and the DP topology run in local cluster mode. This enabled testing of the framework prior to a larger scale deployment.

Machine	Instances	Instance Type	CPU (equivalent)	ECUs	RAM (GB)	Cores
Zookeeper	1	t2.micro	1 x 2.5 GHz Intel Xeon	variable	1	1
Nimbus	1	m4.xlarge	1 x 2.4 GHz Intel Xeon E5-2676v3	13	16	4
Workers	4	c4.2xlarge	1 x 2.9 GHz Intel Xeon E5-2666v3	31	16	8
Workers	8	m4.xlarge	1 x 2.4 GHz Intel Xeon E5-2676v3	13	16	4
TOTAL	14	-	-	241	209	69

Table 2: Summary of the cloud instances deployed to AWS. Here an ECU is an elastic compute unit.

7.2. Cloud Infrastructure

We were awarded compute time upon Amazon's cloud infrastructure via the SKAO-AWS AstroCompute grant programme[7]. This time was used to test the performance and behaviour of the prototype, when scaled beyond a single machine. Using the Amazon Web Services (AWS) console, we provisioned a number of EC2 instances[8]. The provisioned instances are described in Table 2. Note it is difficult to estimate the overall compute capacity possessed by these cloud resources. This is because EC2 instances are deployed on shared hardware, subject to load balancing policies and stress from other EC2 users. Amazon describes the compute capacity of its virtual instances in terms of EC2 Compute Units (ECUs). According to Amazon's documentation, a single ECU corresponds to a 1.0 - 1.2 GHz 2007 Intel Xeon Processor. To map this ECU unit to a meaningful value, consider the 'slowest' (lowest clock speed) Xeon available in 2007, the Xeon E7310. This CPU possesses 4 cores, performs 4 operations per cycle, and has a clock speed of 1.6 GHz. To estimate the FLOPs capability of this processor, we use the formula,

$$\text{FLOPs} = \text{sockets} \cdot \frac{\text{cores}}{\text{sockets}} \cdot \text{clock} \cdot \frac{\text{operations}}{\text{cycle}}. \qquad (10)$$

The Xeon E7310 (1 ECU) is capable of a theoretical throughput of approximately 25.6 GFLOPs, according to both Equation 10 and Intel's own export specifications[9]. The 241 ECUs used during cloud experimentation, therefore correspond to an approximate computational capacity of 6.2 TFLOPs.

8. Evaluation

8.1. Runtime Performance

Topology performance is measured using the auditing nodes. These maintain statistics which are updated after each tuple is processed. Crucially, performance results differ according to the topology configuration used. The configuration describes the number of spouts and bolts in each layer, and the cardinality relationship between them. We adopt a colon delimited notation to describe the configuration, e.g. $1^{1\ldots *} : 2^{1\ldots 1} : 1$. This corresponds to one input spout in layer 1, two processing bolts in layer 2, and a lone bolt in layer 3. The superscript defines the cardinality relation between the current layer and the next. For this example, there is a one-to-many relation between the spout and the bolts in layer 2, and a many-to-one relation between the bolts in layer 2, and the lone bolt in layer 3.

8.2. Filtering Accuracy

The spouts ensure ground truth class labels are known for all candidates entering the topology *a priori*. It is therefore possible to evaluate filtering decisions at any bolt or spout. There are four outcomes for a binary filtering decision, where pulsars are considered positive (+), and non-pulsars negative (-). A **true** negative/positive, is a negative/positive candidate **correctly** filtered. Whilst a **false** negative/positive, is a negative/positive candidate **incorrectly** filtered. It is desirable to have as few false negative/positive outcomes as possible. The outcomes are eventually evaluated using standard metrics such as recall, precision, accuracy and F1 score. These are borrowed from machine learning research[10] [metrics used listed in 59]. Note that we do not compare classification results obtained with the GH-VFDT against existing off-line pulsar classifiers. This is because it is inappropriate to compare off-line and on-line methods in this way - they are intended for fundamentally different processing paradigms.

For the evaluation we use imbalance ratios much higher than those typically observed in the real-world. We do this for a simple reason - if pulsar examples are rare in our test data streams, the prototype will only experience minimal computational load following the classification step (i.e. most candidates will be filtered out here). Since known source matching is a relatively expensive computational operation, we wish to push the prototype to match many more candidates. Hence why we use $c_{\text{ratio}} = 0.05$ or $c_{\text{ratio}} = 0.1$, versus the real-world $c_{\text{ratio}} = 0.0001$ (or worse).

[7] `https://aws.amazon.com/blogs/aws/new-astrocompute-in-the-cloud-grants-program`.

[8] These are virtual machines.

[9] See `http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_7300.pdf`.

[10] We do not use imbalanced metrics here (e.g. the G-Mean or Mathews Correlation Coefficient) as we are not trying to show superior imbalanced class performance, but rather pipeline feasibility).

c_{obs}	Acc.	Recall	F1	t_{avg}(ms)	t_{tot} (s)
100,000	.999	.811	.771	44	38
200,000	.999	.811	.771	53	60
500,000	.999	.811	.771	59	80
1,000,000	.999	.810	.770	58	120
1,500,000	.999	.811	.771	100	225

Table 3: Results for the pipeline run on a local 'cluster' rounded to 3.d.p. Here c_{obs} is the total number of candidates entering the topology, t_{avg}(ms) the time taken on average to process a tuple, and t_{tot} (s) the total time for the pipeline to process all candidates. For these tests a candidate ratio of $c_{\text{ratio}} = 0.05$ was used. See Section 8. for details on how to interpret the configuration used, and Section 8.1 for why such imbalance ratios were used. All results here where obtained for the configuration $1^{1\ldots*} : 4^{1\ldots*} : 11^{1\ldots*} : 4^{1\ldots*} : 4^{1\ldots*} : 4^{1\ldots*} : 4^{1\ldots1} : 1$.

Configuration	Bolts	t_{avg} (ms)	Workers	ECUs
$2^{1\ldots*} : 2^{1\ldots*} : 11^{1\ldots*} : 2^{1\ldots*} : 2^{1\ldots*} : 2^{1\ldots*} : 2^{1\ldots1} : 2$	23	16.503	1	20
$2^{1\ldots*} : 4^{1\ldots*} : 11^{1\ldots*} : 4^{1\ldots*} : 4^{1\ldots*} : 4^{1\ldots*} : 4^{1\ldots1} : 4$	35	21.622	2	40
$2^{1\ldots*} : 8^{1\ldots*} : 11^{1\ldots*} : 8^{1\ldots*} : 8^{1\ldots*} : 8^{1\ldots*} : 8^{1\ldots1} : 8$	59	7.801	4	80
$2^{1\ldots*} : 16^{1\ldots*} : 11^{1\ldots*} : 16^{1\ldots*} : 16^{1\ldots*} : 16^{1\ldots*} : 16^{1\ldots1} : 16$	107	6.891	8	160
$2^{1\ldots*} : 24^{1\ldots*} : 11^{1\ldots*} : 24^{1\ldots*} : 24^{1\ldots*} : 24^{1\ldots*} : 24^{1\ldots1} : 24$	155	2.045	12	240

Table 4: Performance and filtering accuracy results for the prototype pipeline run on a remote AWS cluster processing 1.5 million candidates. Here t_{avg} (ms) is the time taken on average for a tuple to move through the whole pipeline (not counting communication overheads). For these tests a candidate ratio of $c_{\text{ratio}} = 0.1$ was used. See Section 8.1 for details on how to interpret the configuration used, and Section 8.2 for why such imbalance ratios were used.

9. Results

9.1. Local 'Cluster'

The results of local cluster experimentation are given in Tables 3 and 5. The prototype is able to process 1.5 million candidates in well under 600 seconds, as shown in Table 3. It is able to achieve this at a very fast rate, exceeding 1,000 candidates processed per second (up to 6,000 per second during high throughput tests). Thus the prototype is capable of running functionally at SKA scales, on present day commodity hardware[11].

The prototype was functionally effective in local mode, achieving 99% end-to-end accuracy using the global sift algorithm, for 81% pulsar recall. At the bolt level, the performance of global sift is good, but not exceptional. Mistakes made by global sift largely account for pulsars missed by the pipeline. Indeed during isolated bolt-level testing, global sift was found to achieve pulsar recall rates as low as 59.9% with a corresponding accuracy of 99.1%. Lower recall are caused by over-zealous sifting, which happens when the periods of real pulsars fall into the same frequency counting bin as non-pulsars in global sift. Clearly this is a weakness of the approach. Though this can be improved by taking other variables into account when sifting (DM etc). If applied to real SKA data, the prototype would likely miss almost 20% of all pulsars entering the system. However with improvements to sifting and known source matching possible, recall rates can likely be improved with further work.

[11] Achieved on a single laptop, possessing only 70.4 GFLOPs [34] of computational capacity (theoretical max).

Bolt	Acc.	Recall	F1	t_{avg}(ms)
Global Sift	.991	.599	.371	2
ML Classification	.844	.880	.358	2
Source matching	0.999	0.995	.999	7

Table 5: The runtime and filtering performance of bolts tested individually in local cluster mode. Accuracy values are only provided for bolts that filter the data. Here c_{obs} is the total number of candidates entering the bolt, and t_{avg}(ms) the time taken on average to process a tuple. Experiments run using $c_{\text{obs}} = 50,000$, and $c_{\text{ratio}} = 0.05$ (produces many duplicates for sifting).

The individual bolts in the topology appear runtime efficient. Each tuple required on average only a few milliseconds of processing time at each bolt. The runtime per tuple does generally increase as more data is processed as shown in Table 3. The same is true of the total runtime. There is a disparity between the average processing times shown in Table 3, and the bolt processing times in Table 5. The disparity is accounted for by the time taken to transmit each tuple through the network. The values in Table 3 include this transmission time, whilst the values in Table 5 account for only the processing. The difference serves as a reminder that transmission time plays a significant role in the total runtime of a Storm-like system. It also emphasises the importance of running tests outside of local cluster mode, which does not incur the communication overheads experienced in the real-world.

9.2. Cloud Infrastructure

Cloud infrastructure simulation results are presented in Tables 4 and 6. Table 4 shows topology performance according to the *total time* it takes to process a tuple on average. The results in Table 4 show an increase in pro-

cessing time, when initially beginning to scale the topology (when using 1-4 workers). This is followed by a decrease in processing time when more workers are added (increasing the computational power available during execution). Note we do not show filtering accuracy results in these tables. This is because filtering performance does not change between local and remote mode - the bolts and their filters are unchanged. The only difference is that remote mode allows the topology to be scaled to utilise greater computational resources, speeding up processing times and increasing capacity.

The initial increase in processing time observed when beginning to scale the topology is explainable by the results shown in Table 6. These describe the performance of the processing layers, as they are scaled-up (more bolts added) to process the data. When a topology contains only a few instances of each type of bolt, layers reach their processing capacity quickly. This causes an overall increase in processing times due to resource contention. As more bolts are added to the topology, counter-intuitively, the contention is not necessarily reduced. Whilst some bolts will begin to cope with the load when more instances are added, others will not. This happens when resource contention is shifted to another location in the topology, increasing processing times there instead. Note that these results were achieved when running the random 50:50 sifting approach described in Section 9.1. The random sifting approach had to be used instead of our global sift, as global sift is efficient enough to prevent us from stressing the downstream bolts significantly. As random sift propagates far more tuples, it allows us to study the scalability of the topology under more extreme loads.

Contention arises when bolts undertake different proportions of the total computational workload. In the case of the pulsar search topology, known source matching is the most computationally expensive procedure (see Table 6). Adding more bolts to the lower layers of the pulsar search topology, increases the tuple throughput reaching known source matching bolts. Initially the matching bolts cannot meet this demand. Only when at least 8 worker nodes are available, with enough known source matching bolts, does the contention disappear and processing time decrease. This is an important observation. The computational demands of known source matching are surprising.

Table 6 shows which bolts experienced most resource contention. This is indicated by the capacity value (CV),

$$\mathrm{CV} = \frac{(\text{tuples executed} \times \text{avg. execute latency})}{\text{measurement time}}. \quad (11)$$

A value of 1.0 corresponds to a bolt at full capacity. Values greater than 1.0 indicate a bolt over capacity, and less than 1.0 under capacity. When only two bolts are present in each layer of the topology, all bolts are at or near capacity. As the topology is scaled, most bolts begin to become under utilised. The exceptions are the ML classification bolts, and the known source matching bolts.

As more bolt instances are added to each layer of the topology, execution time latency reduces, and total runtime decreases. In some cases the runtime decrease scaled linearly with the number of worker nodes used. However, this is not always the case. There are fluctuations in the results which make it difficult to discern a genuine trend. The overall results shown in Table 4 indicate an improvement in performance scaling sub-linearly with the total number of worker nodes. This impression is based on averaged results, and agrees well with empirical experience. It is therefore likely the most accurate indication of true system scalability. We do not report total runtime, as all tests completed well within the $t_{obs} = 600$ seconds required in the pulsar domain. This suggest our topology can process data in in real-time.

10. Conclusions

A prototype data processing pipeline for pulsar search has been developed. It is capable of on-line and real-time operation. It employs a combination of resource efficient algorithms and optimised selection methods. Together these enable large numbers of pulsar candidates to be filtered very accurately, using limited computational resources.

The performance of the prototype was first assessed on a single commodity laptop. During testing it was able to process 1.5 million pulsar candidates[12], in under 600 seconds. It is therefore functionally capable of processing data fast enough to meet SKA design requirements (exceeds a processing rate of 6,000 candidates per second). The prototype was also deployed to a cloud-based software infrastructure. Here the system was similarly able to process data at SKA scales, using modest computational resources (6.2 TFLOPs of processing power). The runtime performance of the prototype scaled sub-linearly with the number of worker nodes used to execute the processing. Better scaling is likely impeded by the overhead of inter-node communication (i.e. latency and bandwidth restrictions incurred due to network communication).

However, the prototype was designed to favour computational efficiency over filtering accuracy and pulsar recall. This was done to ensure feasible operation at SKA scales, assuming a worst case processing scenario. This trade-off reduced pulsar recall. Thus although filtering accuracy reached 99%, the corresponding pulsar recall rate ranged between 81-88%. This is below the level required for SKA use. There is room to reverse the efficiency-accuracy trade-off, without significantly compromising runtime performance.

[12] The quantity delivered per SKA observation.

Bolt	Instances	Capacity	Execute Latency (ms)	Process latency (ms)
Preprocessing	2	1.984	1.471	1.955
Preprocessing	4	0.048	0.340	0.238
Preprocessing	8	0.012	0.167	0.698
Preprocessing	16	0.008	0.169	0.111
Preprocessing	24	0.008	0.189	0.369
Sift	11	0.039	0.204	0.094
Feature Extraction	2	1.036	1.570	1.488
Feature Extraction	4	0.006	0.176	0.942
Feature Extraction	8	0.006	0.141	0.103
Feature Extraction	16	0.029	0.620	0.205
Feature Extraction	24	0.126	0.595	0.128
ML Classification	2	1.730	2.695	1.669
ML Classification	4	0.002	0.061	6.781
ML Classification	8	1.445	8.884	0.576
ML Classification	16	0.003	0.120	0.694
ML Classification	24	0.074	0.429	0.213
Known Source matching	2	0.967	1.846	1.844
Known Source matching	4	0.298	21.167	6.636
Known Source matching	8	0.700	14.232	16.097
Known Source matching	16	0.271	8.148	5.065
Known Source matching	24	0.277	4.829	4.961

Table 6: The runtime performance of individual bolts using random sift (randomly make sift decision with 50:50 split). Random sift was used to place greater processing load upon downstream bolts in the topology. Here process latency is the time taken to 'ack' a tuple after it is received. Note that 'acking' a tuple involves sending an acknowledgement to the transmitter of a tuple, so it knows it has been received. Execute latency is the time taken for the bolt to complete processing a tuple. Experiments run using $c_{\text{ratio}} = 0.1$. Each experiment (row) repeated 10 times and the average result recorded here.

It would be reasonable to double the resource use of the system, to achieve higher filtering accuracy and pulsar recall.

Finally we emphasise that it was not our intention to suggest we should use so few resources (i.e. a modest 6.2 TFLOPs easily surpassed by a single modern GPU[13]) to run a pulsar search pipeline. Rather we aimed to show that we can do so, using COTS tools at a low price point. A recall rate of 81-88% is promising, but the remaining 12-20% is incredibly difficult to isolate. As the recall rate increases, it becomes increasingly difficult to improve it further. Methods capable of isolating the last few percent will likely be sophisticated, and possibly be accompanied by higher computational runtime costs.

We recommend that future work focus upon developing more sophisticated filters and learning algorithms, as these will likely greatly improve the pulsar recall rate. The algorithms installed at the bolts in the topology should also be studied in greater detail. There is scope to improve their runtime performance, and more crucially, filtering accuracy. We are currently working on a data generator that will assist us in such an investigation.

[13] A NVIDIA Tesla V100 achieves 7.8 TFLOPs (double precision). See `https://www.nvidia.com/en-us/data-center/tesla-v100/`.

11. Acknowledgements

Experimental data was obtained by the High Time Resolution Universe Collaboration. Cloud testing was supported by the SKAO-AWS AstroCompute grant programme. We thank and acknowledge the SKA Time domain team for their support, and all our anonymous reviewers for their helpful feedback.

References

[1] ALFA Pulsar Consortium, 2015, "ALFA Pulsar Studies", on-line, `http://www.naic.edu/alfa/pulsar/`, accessed 06/09/2015.

[2] Allen B. et al., 2013, "The Einstein@Home Search for Radio Pulsars and PSR J2007+2722 Discovery", ApJ, 773. doi:10.1088/0004-637X/773/2/91

[3] Apache Software Foundation, 2018, "Apache Samoa", on-line, `https://samoa.incubator.apache.org/`, accessed 10/10/2018.

[4] Apache Software Foundation, 2015, "Apache Storm", on-line, `http://storm.apache.org`, accessed 22/02/2016.

[5] Apache Software Foundation, 2015c, "S4 Distributed Stream Computing Platform", on-line, `http://incubator.apache.org/s4/`, accessed 22/02/2016.

[6] Apache Software Foundation, 2015d, "What is Samza?", on-line, `http://samza.apache.org`, accessed 22/02/2016.

[7] Apache Software Foundation, 2014, "Welcome to Apache Hadoop", on-line, `http://hadoop.apache.org`, accessed 22/02/2016.

[8] Apache Software Foundation, 2016, "Spark: Lightning-fast cluster computing", on-line, `http://spark.apache.org`, accessed 22/02/2016.

[9] Barsdell B. R. et. al., 2012, "Accelerating incoherent dedispersion", MNRAS, 422(1). doi:10.1111/j.1365-2966.2012.20622.x

[10] Barr E. D., 2014, "Survey for Pulsars and Extra-galactic Radio Bursts", on-line, `http://www3.mpifr-bonn.mpg.de/div/jhs/Program_files/EwanBarrCrete2014.pdf`, accessed 06/09/2015.

[11] Barr E. D. et. al., 2013, MNRAS, 435(3):2234–2245. doi:10.1093/mnras/stt1440

[12] Bates S. D. et al., 2012, MNRAS, 427, 1052. doi:10.1111/j.1365-2966.2012.22042.x

[13] Bifet A., 2015, "Mining Big Data Streams with Apache SAMOA", Proceedings of the 6th International Conference on Mining Ubiquitous and Social Environments - Volume 1521, MUSE'15

[14] Bhattacharyya B., et. al., 2016, "The GMRT high resolution southern sky survey for pulsars and transients. i. survey description and initial discoveries", ApJ, 817(130). doi:10.3847/0004-637X/817/2/130

[15] Bishop C. M., 2006, "Pattern Recognition and Machine Learning", Springer.

[16] Broekema P. C., van Nieuwpoort R. V., and Bal H. E., 2015, *Journal of Instrumentation*, 10(07):C07004. doi:10.1088/1748-0221/10/07/C07004

[17] Carbone P. et al., 2015, "Apache Flink: Stream and batch processing in a single engine", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36 (4).

[18] Chennamangalam J. et al., 2015, "ALFABURST: A realtime fast radio burst monitor for the Arecibo telescope", ArXiv e-prints, astro-ph.IM, arXiv:1511.04132.

[19] Chennamangalam J. et al., 2017, "SETIBURST: A Robotic, Commensal, Realtime Multi-science Backend for the Arecibo Telescope", ApJ Supplement Series, 228, 2. doi:10.3847/1538-4365/228/2/21

[20] Chintapalli S. et al., 2016, "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming", IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). doi:10.1109/IPDPSW.2016.138

[21] Coenen T., 2014, "The LOFAR Pilot Surveys for Pulsars and Fast Radio Transients", A & A, 570(A60). doi:10.1051/0004-6361/201424495

[22] Cordes J. M., 2006, "Arecibo Pulsar Survey Using ALFA. I. Survey Strategy and First Discoveries", ApJ, 637(1):446–455. doi:10.1086/498335

[23] Couturier R., 2013, "Designing scientific applications on GPUs", CRC Press.

[24] Cranmer M. D. et. al., 2017, "Bifrost: a Python/C++ Framework for High-Throughput Stream Processing in Astronomy", ArXiv e-prints, astro-ph.IM, arXiv:11708.00720.

[25] De K. & Gupta Y., 2016, "A real-time coherent dedispersion pipeline for the giant metrewave radio telescope", Experimental Astronomy, 41, 1, p.67–93. doi:10.1007/s10686-015-9476-8

[26] Dewdney P. E., 2013, "SKA1 system baseline design", Technical report, SKA Organization.

[27] Dewdney P. E., 2015, "SKA1 system baseline design v2", Technical report, SKA Organization.

[28] Dimoudi S. and Armour W., 2015, "Pulsar acceleration searches on the GPU for the square kilometre array", In *25th annual ADASS conference*, p.6.

[29] Dubey R., 2008, "Introduction to embedded system design using field programmable gate arrays", Springer Science & Business Media.

[30] Duda R. O., Hart P. E., and Stork D. G., 2000, "Pattern Classification", Wiley-Interscience, 2nd Edition.

[31] Eatough R. P., Molkenthin N., Kramer M., Noutsos A., Keith M. J., Stappers B. W., Lyne A. G., 2010, MNRAS, 407, 2443. doi:10.1111/j.1365-2966.2010.17082.x

[32] Gorawski M., Gorawska A. and Pasterak K., 2014, "A Survey of Data Stream Processing Tools", Information Sciences and Systems 2014, p.295-303, Springer.

[33] Hessels J. W. T. et. al., 2006, "A Radio Pulsar Spinning at 716 Hz", American Astronomical Society Meeting Abstracts, 207.

[34] Intel Corporation, 2013, "Intel®Core i7-2700 Mobile Processor Series", on-line, `http://www.intel.com/content/dam/support/us/en/documents/processors/corei7/sb/core_i7-2700_m.pdf`, accessed 28/04/2016.

[35] Jain A. and Nalya A., 2014, "Learning Storm", Packt Publishing.

[36] Jankowski M., Pathirana P., and Allen S. T., 2015, "Storm Applied: Strategies for Real-time Event Processing", Manning.

[37] Jones M. T., 2013, "Process real-time big data with twitter storm", IBM Technical Library.

[38] Karako-Argaman C. et. al., 2015, "Discovery and Follow-up of Rotating Radio Transients with the Green Bank and LOFAR Telescopes", ApJ, 809, 1. doi:10.1088/0004-637X/809/1/67

[39] Karastergiou A. et. al., 2015, "Limits on fast radio bursts at 145 MHz with Artemis, a real-time software backend", MNRAS, 452(2):1254–1262. doi:10.1093/mnras/stv1306

[40] Keane E. F., 2016, "The host galaxy of a fast radio burst", Nature 530, 453–456. doi:10.1038/nature17140

[41] Keith M. J. et. al., 2010, "Discovery of 28 pulsars using new techniques for sorting pulsar candidates", MNRAS, 395(2):837–846. doi:10.1111/j.1365-2966.2009.14543.x

[42] Keith M. J. et. al., 2010, "The High Time Resolution Universe Pulsar Survey - I. System Configuration and Initial Discoveries", MNRAS, 409(2):619–627. doi:10.1111/j.1365-2966.2010.17325.x

[43] Keith M. J., 2016, "Pulsar hunter", on-line, `http://www.pulsarastronomy.net/wiki/Software/PulsarHunter`, accessed 22/02/2016.

[44] Kilts S., 2007, "Advanced FPGA design: architecture, implementation, and optimization", John Wiley & Sons.

[45] Kleppmann S. and Kreps J., 2015, "Kafka, Samza and the Unix Philosophy of Distributed Data", IEEE Data Eng. Bull., 38(4), pp.4-14.

[46] Knuth D. E., 1970, "Optimum binary search trees", Acta Informatica, 1(1), pp.14-25. doi:10.1007/BF00264289

[47] Kulkarni S. et. al., 2015, "Twitter heron: Stream processing at scale", In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, p.239–250. doi:10.1145/2723372.2742788

[48] Law C. J. et. al., 2015, "A Millisecond Interferometric Search for Fast Radio Bursts with the Very Large Array, ApJ, 807(1). doi:10.1088/0004-637X/807/1/16

[49] Lazarus P., 2012, "The PALFA Survey: Going to great depths to find radio pulsars", In *Neutron Stars and Pulsars: Challenges and Opportunities after 80 years*, volume 8 of *Proceedings of the International Astronomical Union Symposium S291*, p.53–56.

[50] Lee K. J. et al., 2013, MNRAS, 433, 688. doi:10.1093/mnras/stt758
Lopez:2016:ll

[51] Lopez M. A. et. al., 2016, "A Performance Comparison of Open-Source Stream Processing Platforms", IEEE Global Communications Conference (GLOBECOM).

[52] Lorimer D. R., 2016, "Sigproc", on-line, `http://sigproc.sourceforge.net`, accessed 22/02/2016.

[53] Lorimer D., Kramer, M., 2006, Cambridge Univ. Press.

[54] Lorimer D. R., 2007, "A Bright Millisecond Radio Burst of Extragalactic Origin", *Science*, 318(5851):777–780. doi:10.1126/science.1147532

[55] Lyon R. J. et. al., 2013, "A Study on Classification in Imbalanced and Partially-Labelled Data Streams", *IEEE International Conference on Systems, Man, and Cybernetics*, p.1506–1511. doi:10.1109/SMC.2013.260

[56] Lyon R. J. et. al., 2014, "Hellinger Distance Trees for Imbalanced Streams", In *22nd IEEE International Conference on Pattern Recognition*, p.1969–1974. doi:10.1109/ICPR.2014.344

[57] Lyon R. J., 2015, "HTRU2", on-line, doi:10.6084/m9.figshare.3080389.v1

[58] Lyon R. J., 2015, " Why Are Pulsars Hard to Find?", PhD thesis, University of Manchester, School of Computer Science.

[59] Lyon R. J. et. al., 2016, "Fifty Years of Pulsar Candidate Selection: From simple filters to a new principled real-time classification approach", MNRAS, 459 (1):1104-1123. doi:10.1093/mnras/stw656

[60] Lyon R. J., 2017, "Supporting Material: A Big Data Pipeline for High Volume Scientific Data Streams", on-line, doi:10.5281/zenodo.1116302

[61] Magro A. et. al., 2011, "Real-time, fast radio transient searches with GPU de-dispersion", MNRAS, 417(4):2642–2650 doi:10.1111/j.1365-2966.2011.19426.x

[62] Manchester R. N. et. al., 2005, "The Australia Telescope National Facility Pulsar Catalogue", ApJ, 129, 4, doi:10.1086/428488, `http://www.atnf.csiro.au/people/pulsar/psrcat/`, accessed 28/04/2016.

[63] McLaughlin M. A., 2009, "Rotating radio transients", In W. Becker, editor, *Neutron Stars and Pulsars*, p.41–66. Springer Berlin Heidelberg.

[64] Morello V., Barr E. D., Bailes M., Flynn C. M., Keane E. F., van Straten W., 2014, MNRAS, 443, 1651. doi:10.1093/mnras/stu1188

[65] Naidu A. et. al., 2015, "PONDER - a real time software backend for pulsar and IPS observations at the ooty radio telescope", *Experimental Astronomy*, 39(2):319–341. doi:10.1007/s10686-015-9450-5

[66] Neumeyer L. et. al., 2010, "S4: Distributed Stream Computing Platform", IEEE International Conference on Data Mining Workshops.

[67] Nijboer R. et. al., 2015, "PDR.05 parametric models of SDP compute requirements", Technical report, SKA Organization.

[68] Ng C., 2012, "Conducting the deepest all-sky pulsar survey ever: the all-sky High Time Resolution Universe survey", In *Neutron Stars and Pulsars: Challenges and Opportunities after 80 years*, volume 8 of *Proceedings of the International Astronomical Union Symposium S291*, p.53–56.

[69] Petroff E. et. al., 2015, "A real-time fast radio burst: polarization detection and multiwavelength follow-up", MNRAS, 447(1):246–255. doi:10.1093/mnras/stu2419

[70] Ransom S., 2016, "Presto", on-line, `http://www.cv.nrao.edu/~sransom/presto/`, accessed 22/02/2016.

[71] Russell S. and Norvig P., 2009, "Artificial Intelligence: A Modern Approach", Prentice Hall, 3rd Edition.

[72] Sclocco A. et. al., 2015, "Finding pulsars in real-time", In *2015 IEEE 11th International Conference on e-Science*, p.98–107. doi:10.1109/eScience.2015.11

[73] Shree R. et. al., 2017, "KAFKA: The modern platform for data management and analysis in big data domain", in Proceedings of the 2nd International Conference on Telecommunication and Networks (TEL-NET).

[74] Smits R. et. al., 2009, A & A, 493(3):1161–1170. doi:10.1051/0004-6361:200810383

[75] Stovall K., 2013, "Large scale pulsar surveys, new pulsar discoveries, and the observability of pulsar beams strongly bent by the Sag. A* black hole", PhD Thesis, The University of Texas at San Antonio.

[76] Tan C. M. et. al., 2018, MNRAS, 474(11):4571-4583. doi:doi.org/10.1093/mnras/stx3047

[77] Tan G. H. et. al., 2015, "The square kilometre array baseline design v2.0", In *1st URSI Atlantic Radio Science Conference*. doi:10.1109/URSI-AT-RASC.2015.7303195

[78] Thompson D. R. et. al., 2011, "Semi-supervised eigenbasis novelty detection, and application to radio transients", In *NASA Conference on Intelligent Data Understanding*, p.118–128.

[79] Thornton D., 2013, "The High Time Resolution Radio Sky", PhD thesis, University of Manchester.

[80] Thulasiraman K. & Swamy M. N. S., 1992, "Graphs: Theory and Algorithms", John Wiley & Sons, Inc. doi:10.1002/9781118033104.indsub

[81] Toshniwal A. et. al., 2014, "Storm@twitter", In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, p.147–156, ACM. doi:10.1145/2588555.2595641

[82] van Heerden E. et. al., 2014, "New approaches for the real-time detection of binary pulsars with the square kilometre array", In *General Assembly and Scientific Symposium XXXIth URSI*, p.1–4.

[83] van Nieuwpoort R. V., 2016, "Towards exascale real-time RFI mitigation", in *2016 Radio Frequency Interference (RFI)*, p.69–74. doi:10.1109/RFINT.2016.7833534

[84] van Straten W. & Bailes M., 2011, "DSPSR: Digital Signal Processing Software for Pulsar Astronomy", PASA, 28, 1. doi:10.1071/AS10021

[85] Vanderbauwhede W. and Benkrid K., 2013, "High-Performance Computing Using FPGAs", Springer.

[86] Wang H. and Sinnen O., 2015, "FPGA based acceleration of FDAS module for pulsar search", In *International Conference on Field Programmable Technology (FPT)*, p.240–243. doi:10.1109/FPT.2015.7393158

[87] Weiler A., Grossniklaus M., and Scholl M. H., 2015, "Run-time and task-based performance of event detection techniques for twitter", In *Advanced Information Systems Engineering: 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, p.35–49. doi:10.1007/978-3-319-19069-3_3

[88] White T., 2012, "Hadoop: The definitive guide", O'Reilly Media.

[89] Woods R. et. al., 2008, "FPGA-based Implementation of Signal Processing Systems", Wiley Publishing.

[90] Zhu W. W. et al., 2014, "Searching for Pulsars Using Image Pattern Recognition", ApJ, 781, 2. doi:10.1088/0004-637X/781/2/117

[91] Zaharia M. et al., 2013, "Discretized streams: Fault-tolerant streaming computation at scale", Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, p.423-438.

[92] Zaharia M. et al., 2014, "Apache Spark: A Unified Engine for Big Data Processing", Commun. ACM, 59, 11, p.56-65.