# An Infrastructure to Store and Analyse Seismic Data as Suffix Trees

**MSc Data Analytics - Project Report**

Tom Taylor

September 2017

*This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the programme Handbook and the College website.*

# Contents

# 1 Introduction

## 1.1 Abstract

The purpose of this project is to develop an infrastructure and tool set for converting raw seismic time series data into a searchable string using SAX (**S**ymbolic **A**ggregate appro**X**imation) and then to store this data as a suffix tree for fast searching and analysis. An interface will then be developed to enable the searching of these suffix trees and provide visualisation of the data. Primarily this will be with the aim of being able to identify the start of an event with reasonable accuracy. Additionally it could ideally be used to search for similar patterns over time or between stations after an event.

## 1.2 Objectives

1. To facilitate the importing and storage of raw seismic data and associated meta-data
2. To calculate and store SAX strings of a whole observation period or an event
3. To be able to determine the onset of an event in an observation
4. To provide graphical representations of the analysis
5. To provide a web based user interface for all of the above

# 2 Background

## 2.1 Seismic Waves

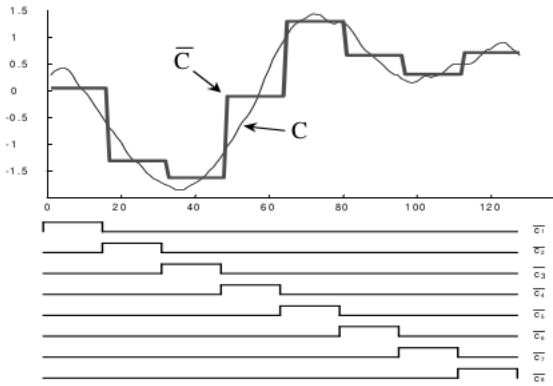
Seismic waves take on two main forms, body waves and surface waves. Body waves are those that travel through the interior of the earth and are the fastest travelling. The body waves are comprised of **P** (primary) waves which are compressional waves, travel fastest and thus arrive first. **S** (secondary) waves are shear waves and travel more slowly, thus arrive later. The separation between the phases is related to the distance of the earthquake and the local velocity structure. The surface waves travel only along the earth's crust and, as they are confined to shallow depths where seismic velocities are slow, will normally arrive much later than the body waves.

A seismic station records movement over three axis: vertical (**z**) alongside horizontal in terms of north-south (**n**) and east-west (**e**). Due to seismic velocities generally increasing with depth, P waves arrive at a seismic station close to the vertical axis. As a result, P waves can be measured as a simple metric of displacement along the Z axis. S waves follow similar ray paths, but have their particle motion perpendicular to the direction of propagation. As a result, they manifest on a seismogram as movement on both the n and e axis. The geometry of the fault tends to have a bearing on the orientation of the displacement so the two horizontal axis of movement cannot be easily combined in to a single metric for time series analysis.

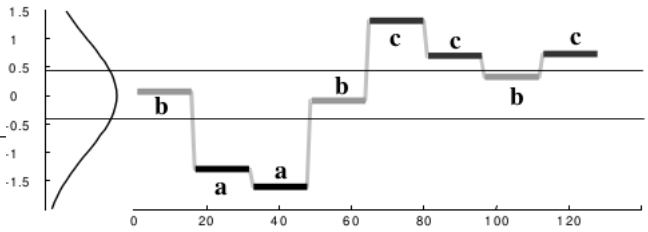
## 2.2 Symbolic Aggregate Approximation (SAX)

SAX (**S**ymbolic **A**ggregate appro**X**imation) (Lin et al., 2003) is a technique where by a single dimension of a time series is reduced to a string of symbols for pattern matching. The technique involves first transforming the normalised time-series in to a Piecewise Aggregate Approximation (**PAA**) which is then represented by a fixed number of symbols. The normalisation technique is *Z-normalisation* which is discussed in section 4.1.

For the PAA, the data is first divided into equal sized time frames (see diagram below), then the mean deviation from zero of each frame is calculated. An appropriate number of breakpoints symmetrical along the x-axis are created so that they follow a Gaussian distribution and a symbol assigned to each range between the breakpoints. Then for each frame, a symbol is assigned based on which range the mean falls in to. The symbols assigned to each frame are then concatenated in to a string and it is this string that gives the SAX representation of that data. The width of the time frame and the number of discrete regions would be two parameters passed to this process alongside the data.




(a) Calculating PAA




(b) Diagrams showing PAA to SAX on a time-series (taken from Lin et al. (2003))

## 2.3 Suffix Trees

Suffix trees (Weiner, 1973) are a representation of a string designed for performing search related algorithms. They are based on a suffix trie of a given string **T**. The trie is constructed using all of the suffixes of **T** with a terminator character not used in the alphabet appended to the end of **T**. The terminator is deemed to be lexicographically of lower priority than all of the other characters to ensure that a prefix of a string would appear before that string (such as in *as* before *ash*). In the following example, the suffixes of *T: abaaba* would produce the following suffixes with **$** applied as a terminator.

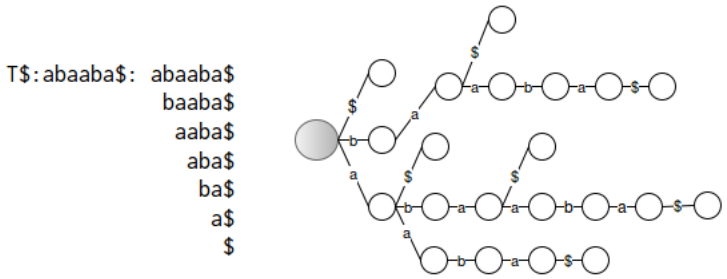



Figure 2: Example Suffix Trie

This allows for efficient searching and counting substrings of **T** by following the trie from the root node. Each substring **S** of **T** is a prefix of a path starting at the root. It allows for counting occurrences of the substring by finding the number of leaf nodes that can be reached from the end of the substring. This format is not particularly space efficient as it has an upper bound storage in the order of $O(n^2)$.

A far more efficient way to store and query the trie is to store it as Suffix Trees. In order to produce a suffix tree from a trie, the non-branching nodes are firstly reduced (or coalesced) in to a single edge. The original string **T** is then stored along side the tree

and the labels further reduced to a starting position and offset within **T**. The leaf nodes become labels to the offset of the suffix in the string. This reduces the upper bound storage to $O(n)$ and results in a tree for our example **T** as seen in section 2.3.

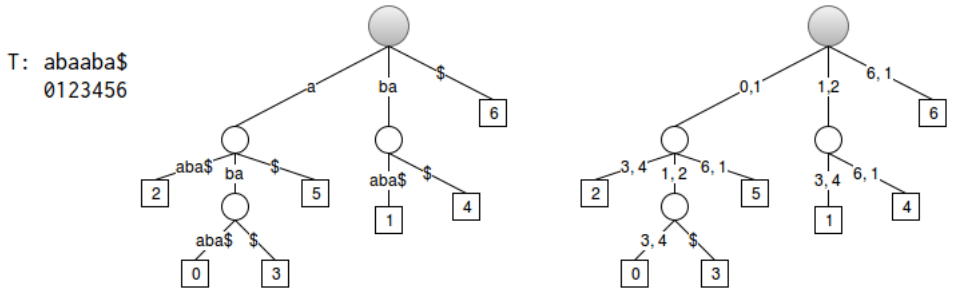



Figure 3: Example Suffix Tree (edges shown as text and as positions/offsets)

This technique of building the tree is considered naive though as it is again very inefficient (being computationally in the order of $O(n^3)$). It is far more desirable to achieve time-linear *O(n)* construction of the tree. An example of this is the Ukkonens Algorithm (Ukkone, 1995). The algorithm is too long for inclusion but effectively starts with the implicit suffix tree of a string of length 1 and conditionally extends the tree on each iteration of adding a character.

At Birkbeck, an infrastructure has been developed to load and query Suffix Trees (Harris et al., 2016). At a high level it is based on language models but should be extendable to time series by the use of SAX. There is current work in progress to utilise external memory (in this case Solid State Drives) to back the loaded Suffix Trees. This would massively increase the maximum size of a tree to be queryable. The current libraries to utilise this are written in Python however there is currently a refactor happening to C due to Python not being particularly computationally efficient because of its interpreted nature. The trees in this implementation are stored as k-truncated trees for practical reasons.

# 3 Plan

The following sections follow the objectives as defined in section 1.2

## 3.1 Importing and Storing Data

Initially the observation data is available as SAC or Miniseed files. These contain the raw observations along with metadata about the station that recorded the data. This data can be parsed with the Obspy Python library (Beyreuther et al., 2010). Use of this library produces an Python object allowing simplified access to the data and metadata from the file. The library also contains many methods for post-processing operations such as signal filtering and commonly used seismic analysis tools.

In the project, this object will be wrapped in to a Data Access Object (DAO). This is to add to or simplify many of the commonly used operations such as down-sampling, time slicing and passing through a bandpass filter to remove high and low frequencies.

For simplicity when working with many files, a service will be written to extract the metadata and store it in a database and the raw observation file will be stored in an object store for later retrieval.

## 3.2 Generate SAX Strings

A library and associated service will be written to carry out the normalisation of data, and to calculate PAA and to generate and return the SAX strings. In itself, this service would not persist the output but return it to the caller to either be rendered or stored.

## 3.3 Event Detection

Event detection will call on data provided by the SAX service and apply rules or heuristics to determine the onset (and potentially the duration) of events. This information will be persisted in a database along with the observation metadata.

## 3.4 Graphical Representation

Rendering of visuals will be rendered client-side in a browser. One or more Javascript libraries will be utilised to show the various stages of the processing and results.

### 3.5 User Interface

An HTML/Javascript interface will be rendered from user facing service utilising the other services and passing data back and forth using HTTP requests and JSON payloads.

The interface will facilitate the importing and storing of data, searching and rendering results. It will also allow for the submission of deferred or long running tasks.

# 4 Data Processing

Due to the widely varying types and models of instrument deployed, the values reported from a Seismic Station are often relative only to themselves. This means while the profile of an event could be established from the seismogram, amplitudes are potentially meaningless when being compared between stations. It is also not possible to directly infer the strength of a quake from a seismogram without empirical evidence from previous events.

## 4.1 Normalisation

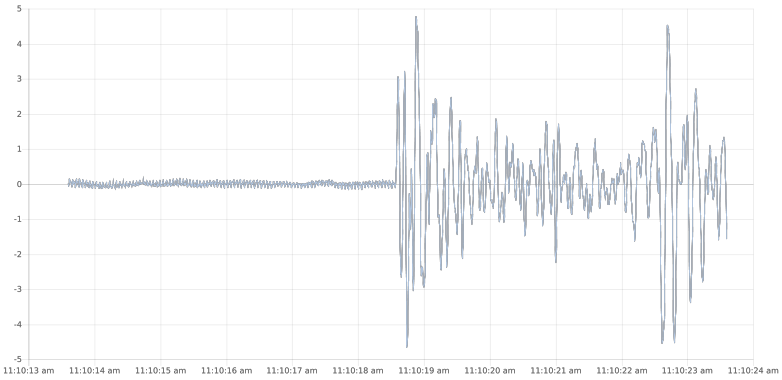
The preprocessing technique for normalisation suggested for SAX (Lin et al., 2003) is *Z-normalisation*. That is to normalise the data so that it has a mean ($\bar{x}$) of zero and a standard deviation ($\sigma$) of one. This is achieved by subtracting the mean from each value and then dividing by the standard deviation.

$$z = \frac{x - \bar{x}}{\sigma} \tag{1}$$

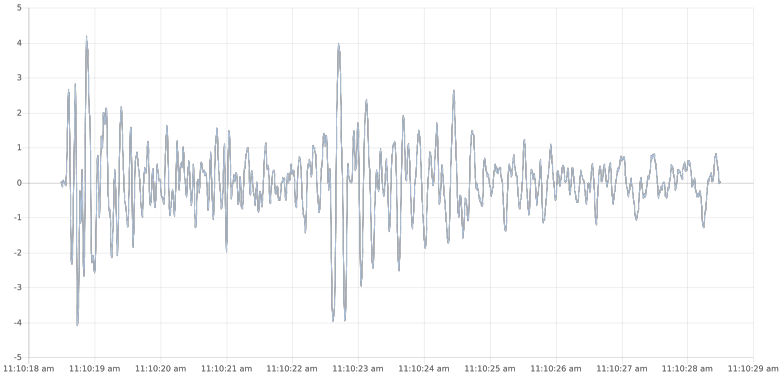
In a closed set of values, this technique often works perfectly fine. Unfortunately in this case, the vast majority of data points on seismic recordings are made up of background noise that are often indistinguishable from the events themselves apart from by amplitude. This means that if background noise is included in the sample from which the mean and standard deviation are taken, then values during the actual event will be over amplified and therefore unsuitable for *SAX* processing.

A further problem introduced by unsupervised normalisation is the tailing off of events. In the same way that introducing otherwise quiet periods amplifies the peaks of the event, so does including too much of the tail.

In view of these issues, and because *z-normalisation* is an important part of *SAX processing*, it is essential that only significant data be presented to be normalised for analysis. To meet this requirement, it is important that the start of the event be relatively accurately estimated and only the first few seconds are treated. This is also important because we are only interested in profiling *P-waves*. The *S-waves* following a few seconds later interfere with the Z-axis recordings. This is demonstrated in fig. 4 where if earlier background noise is included in the sample (a) the peaks are normalised a little over 4 where if only the event is included, the peaks are nearer to 5. When looking at longer time ranges, this effect is amplified significantly rendering the event data meaningless.




(a) With background noise




(b) Without background noise

Figure 4: Comparison of normalisation with and without background

## 4.2 PAA & SAX

As described in the background section section 2.2 on SAX, there are two steps. PAA (Peicewise Aggregate Approximation) is applied before symbols are calculated based on equal breakpoints following a Gaussian distribution.

### 4.2.1 PAA

A Python class was written to be callable to convert a Dataframe into an array of the aggregated values. To reduce the number of steps involved in using the class, it performs the normalisation step unless told not to.

```
import numpy as np
import pandas as pd


class PaaError(Exception):
    pass
```

```
class Paa(object):
    def __init__(self, series=pd.Series, normalise=True):
        """
        Prepare a PAA (Piecewise Aggregate Approximation) object to calculate
        PAA of a given dataset.  Will perform z-normalisation by default on
        data     before interpolating linearly to an interval of 1ms.

        Args:
            series (DataFrame): pandas DataFrame with time as index
            normalise (bool): Whether or not to normalise
        """
        if not isinstance(series, pd.Series):
            raise PaaError("series␣should␣be␣a␣pandas␣Series")
        series = series.resample("1L", how="mean").interpolate(method="time")
        if normalise:
            std = np.std(series)
            mean = np.mean(series)
            series = (series - mean) / std
        self.series = series

    def __call__(self, window=int):
        """
        Return a PAA of the DataFrame

        Args:
            window (int): Number of milliseconds in window

        Returns:
            pandas.Series
        """
        if not isinstance(window, int):
            raise PaaError("Window␣should␣be␣an␣integer")
        df = self.series.copy()
        return df.resample("{}L".format(window)).mean().interpolate(method="
            time")
```

When the class is first instantiated, a copy of the series is stored locally in the object. Unless the normalise parameter is explicitly set to false, the numpy library is used to calculate the mean and standard deviation of the series and then $(x - \bar{x})/\sigma$ is calculated for the whole series as described in section 4.1.

This returns a callable object with the window size (in milliseconds) as a parameter. When called, the object uses the resample feature of Pandas to return a series of mean values. It should be defined and called as follows:

```
p = Paa(series=d)  # where d is a Pandas dataframe with a time index
paa_out = p(50)        # performs PAA on d with a window size of 50ms
```

### 4.2.2 SAX

Similar to the PAA class, the SAX class was written to produce a callable object.

```
from scipy.stats import norm
import numpy as np
from pandas import Series


class SaxError(Exception):
    pass


class Sax(object):
    def __init__(self, paa=Series):
        """
        Provides a generator for SAX data from PAA
        Args:
            paa (pd.Series): result from calling Paa
        """
        if not isinstance(paa, Series):
            raise SaxError("paa should be a pandas.Series, got {}".format(
                type(paa)))
        self.paa = paa

    def __call__(self, alphabet=str):
        """
        Generate SAX string from PAA as a pandas.Series

        Args:
            alphabet (str): alphabet for SAX

        yields:
            str
        """
        if not isinstance(alphabet, str):
            raise SaxError("alphabet should be a str, got {}".format(type(
                alphabet)))
        # Generate gaussian breakpoints
        thresholds = norm.ppf(
            np.linspace(1 / len(alphabet), 1 - 1 / len(alphabet), len(
                alphabet) - 1)
        )
        for i in self.paa:
            yield alphabet[np.searchsorted(thresholds, i)]
```

On instantiation, copy of the original series is stored in the object with no additional pre-processing. The object is then called with the desired alphabet passed as the only parameter. Then length of the string is used to determine the number of breakpoints to calculate against a normal distribution and these are stored as the *thresholds*. A Python generator is then returned that uses the numpy *searchsorted* method to establish between which breakpoints a value falls and then return the corresponding character.

The generator can then be used to iterate over the values one at a time by the calling function. A simple use to print the characters is shown below:

```
s = Sax(paa_out)           # instantiate a callable Sax object
for val in s("abcdefg"):   # iterate over the object
    print(val, end="")     # print each value (with no newline)
```

## 4.3 Frequency Analysis

Another option for analysing the stream was to disregard the amplitudes and concentrate on the frequency domain. That is to convert to a function of frequency against time.

Many of the frequencies that occur within the streams are considered noise, especially those of a higher frequency that are likely caused by wind.

# 5 Event Detection

The primary intent of the project was to be able to use the data generated byt the *SAX* algorithm to detect events in a measurement over a period of time. One existing common technique for doing this is STA/LTA (Short Term Average/Long Term Average) (Bormann, 2012). STA/LTA is a sliding window technique where the mean of the absolute amplitudes are compared ofver two different sized windows. If the STA exceeds the LTA by a preset threshold then an event is triggered, if it drops below another threshold the trigger is ended. The event is considered the duration between these two triggers.

In order to evaluate a SAX based detection algorithm, an implementation of STA/LTA was required for comparison Traditionally STA/LTA thresholds are set manually by experienced seismologists, in this case however, the author does not have the required experience to set these values so an adaptive method was required.

## 5.1 Adaptive STA/LTA Algorithm

*N.B. This implementation has only been tested on a relatively small sample set and is not recommended for real-world use*

The algorithm works by iterating through the data on two sliding windows calculating a short and long term mean of the absolute values of the amplitude.

$l$ = number of datapoints in LTA
$s$ = number of datapoints in STA

$$\overline{LTA} = \sum_{i=1}^{l} \frac{|v(t-i)|}{l} \tag{2}$$

$$\overline{STA} = \sum_{i=1}^{s} \frac{|v(t-i)|}{s} \tag{3}$$

For each iteration, the STA is compared to a number of standard deviations (typically 3) from the LTA, if it exceeds this value, the event is considered triggered on. The current value of the LTA is then stored.

The iteration then continues calculating STA with the triggered value set to true until the STA drops below the original LTA value when the event was triggered.

If the event duration is above a pre-set threshold (typically 5s) then the event is recorded, if not the it is discarded as a probable spike.

One downside to this approach is that it cannot normally detect an event within the period of the LTA from the start of the observation window. It also struggles to detect an event when there is significant background noise.

## 5.2 SAX Detection Algorithm

The *SAX Detection Algorithm* works by firstly calculating a SAX string for the whole observation. It requires a minimum *trigger on* length in milliseconds (to eliminate spikes), a quiet duration (again in milliseconds) and a distance from the centre value of the alphabet used to produce the SAX string. This string is then iterated over one character at a time filling a ring buffer that is at least the length of the *trigger on* or the *quiet duration* that is used to establish whether trigger requirements are met. It is effectively an implementation of Step Detection using the breakpoints defined in the SAX process.

```
import types
from collections import deque

from .exceptions import DetectorError


def distance(alphabet, a, b):
    """
    Calculates the distance from the centre value in the alphabet of a given
        s

    Args:
        alphabet (str): alphabet to search
        a (str): First character
        b (str): Second character

    Returns:
        int: distance from centre value
    """
    if a not in alphabet:
        raise DetectorError("{} was not found in {}".format(a, alphabet))
    if b not in alphabet:
        raise DetectorError("{} was not found in {}".format(b, alphabet))
    return abs(alphabet.find(a) - alphabet.find(b))


def near_centre(alphabet, centre, max, s):
    """
    Returns whether a character is less than max from the centre value of a
        string

    Args:
        alphabet (str): full alphabet to compare from
        centre (str): centre value
        max (int): maximum distance to consider
```

```
        s (str): search character

    Returns:
        bool
    """
    return s == centre or distance(alphabet, centre, s) <= max


def sax_detect(stream, alphabet, paa_int, off_threshold=5000, min_len=5000):
    """

    Args:
        stream (types.GeneratorType): Generator of SAX string
        alphabet (str): Alphabet used to create SAX string
        paa_int (int): PAA Window size in ms
        off_threshold: Quiet period to consider event finished
        min_len: Minimum length of an event to yield

    Yields:
        (start_ms, end_ms)
    """
    if not isinstance(stream, types.GeneratorType):
        raise DetectorError(
            "SaxDetect␣stream␣expects␣a␣generator,␣got␣{}".format(type(stream
                )))
    if not isinstance(alphabet, str):
        raise DetectorError(
            "SaxDetect␣alphabet␣expects␣a␣str,␣got␣{}".format(type(alphabet))
                )
    if not len(alphabet) % 2 == 1:
        raise DetectorError(
            "SaxDetect␣requires␣an␣odd␣length␣of␣alphabet␣to␣have␣a␣centre")
    # Convert the next two values from ms to number of elements
    min_len = int(min_len / paa_int)
    off_threshold = int(off_threshold / paa_int)
    # Ring buffer for looking back at recent values
    buffer = deque([], maxlen=off_threshold)
    # Centre value to calculate distance from
    centre = alphabet[int(len(alphabet) / 2)]
    i = 0  # current pos
    t_on = 0  # trigger on value
    triggered = False
    for s in stream:
        buffer.appendleft(s)
        if not triggered:
            if not near_centre(alphabet, centre, 1, s):
                triggered = True
                t_on = i
        elif triggered:
            if near_centre(alphabet, centre, 1, s):
                for j in range(1, off_threshold):
                    if not near_centre(alphabet, centre, 1, buffer[j]):
                        break
                else:
```

```
            triggered = False
            if i - t_on >= min_len:
                yield t_on * paa_int, (i - off_threshold) * paa_int
i += 1
```

# 6 Application Design

## 6.1 Discrete Services

The application was decided to be split in to discrete components with an all of the data processing bundled together behind an API exposed via HTTP with mostly JSON payloads (the observation data is transferred and stored in its original binary format). The interface is a separate service that is mostly an HTML and Javascript powered application that speaks to the API on the HTTP/JSON interface. A third instance of the application runs to carry out long running or deferred tasks and is communicated with via Redis as a message queue. Additionally data persistence services (see section 6.3) run separately.

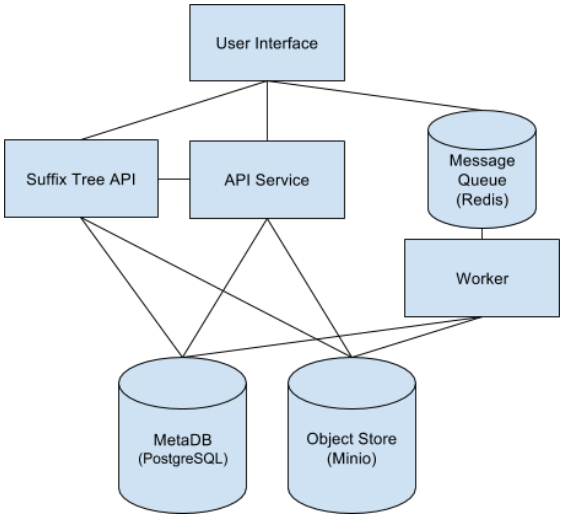



Figure 5: Application Architecture

## 6.2 Data processing API

As previously mentioned, the data processing functions (with the exception of the Suffix Tree code) are bundled in to a single API process broken in to separate namespaces for the different parts.

The *observations* namespace used to manipulate observation files. It supports a REST like interface for CRUD (Create, Read, Update and Delete) operations and RPC (Remote Procedure Call) for running event detection and retrieving results.

The *SAX* namespace is exposed via a JSON based HTTP API. Its purpose is to perform PAA and SAX operations on a given observation or event and return data for rendering visualisations as well as the produced string from the SAX calculation.

The HTTP/JSON APIs were written using Flask-Restplus (Haustant, 2016), a Python framework based on Flask that allows for request and response definition using Python Decorators around classes and methods. The framework allows for dynamic generation of a swagger.json and provides a Swagger interface. Swagger is a commonly used standard for defining APIs. It also provides a user friendly HTML based interface for testing methods and calls during development and also doubles as API documentation. An example of Swagger used on the Suffix Tree API is shown in fig. 6.

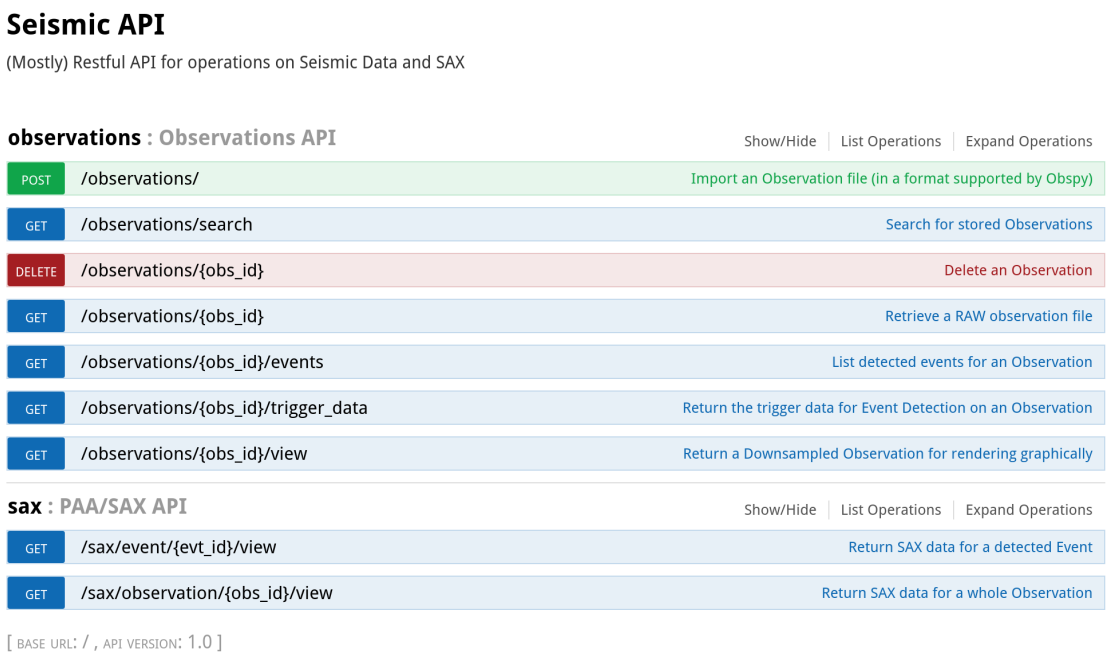



Figure 6: Swagger interface for Data Processing API

### 6.2.1 Observations API Namespace

### 6.2.2 SAX API Namespace

## 6.3 Data Persistence

For the persistence of Metadata about observations, detected events and Suffix Trees, PostgreSQL was selected. PostgreSQL is a performant and mature open-source Relational Database Management System (RDBMS). Being a fully featured RDBMS means it brings ACID guarantees for data resilience and transaction management.

For the persistence of Binary objects (such as Raw Observation Files and Suffix Trees), Minio was selected. Minio is an open-source object storage application written in Golang designed to emulate the abilities of Amazons Simple Storage Service (S3). It allows for

arbitrary binary objects to be stored and retrieved remotely from *buckets* of grouped resources. In each bucket, an object has a unique name that can also emulate a filesystem path (e.g. *bucket*/somedir/somefile).

A wrapper class was written around Minio that provides Get, Put and Delete operations. This interface was intentionally left simple so that it could be replaced easily by another object or file store.

## 6.4 User Interface

The user entry point to the application is served by the *interface* service. This provides a web based interface to the various backend components and rendering of graphs as well as acting as a reverse proxy to the various APIs behind the application.

As mentioned previously, the interface to the application is web-based. This serves two purposes; firstly it provides a unified experience across operating systems and secondly some of the processing can be memory and CPU intensive so is better suited for running on server hardware. It also facilitates the potential sharing of data between users.

The interface is served using Flask for Python. Flask allows for dynamic request handling by converting URL paths directly in to function calls in Python. It also supports HTML templating via Jinja2. An module was written to act as a reverse proxy to expose various sections of the backend APIs to the browser.

The view and control components of the interface were written in HTML (using Twitter's Bootstrap) and Javascript (using AngularJS, Charts.js and c3.js). AngularJS allows for two way data binding between the Broswers DOM and elements such as form inputs and renders with minimal additional code. It also provides a mechanism for sending requests to the APIs exposed by the aforementioned Proxy module. Chart.js and c3.js are two open source Javascript charting libraries that utilise features included in HTML5 used in this case for rendering observation data.

## 6.5 Deferred & Batch Jobs

For the running of deferred and batch jobs, Celery (another Python based project) was selected. It uses a message queue for receiving and dispatching jobs. Celery allows for the definition of jobs as functions which can then be imported by other processes (in this case, mostly the user interface). The application uses Redis as the message queue. It also allows for the chaining together of tasks to ensure order (e.g. event detection before SAX analysis) and the passing of results on to the next function in a functional programming style.

## 6.6 Suffix API

The Suffix API runs as a separate service as the majority of the code base is taken from previous work on suffix trees at Birkbeck (Harris et al., 2016). This was done mainly to separate it from the original work on this project. The only additions were an HTTP API written in the same style as the Data Processing API so that it could be interacted with. The API operations are summarised in fig. 7.

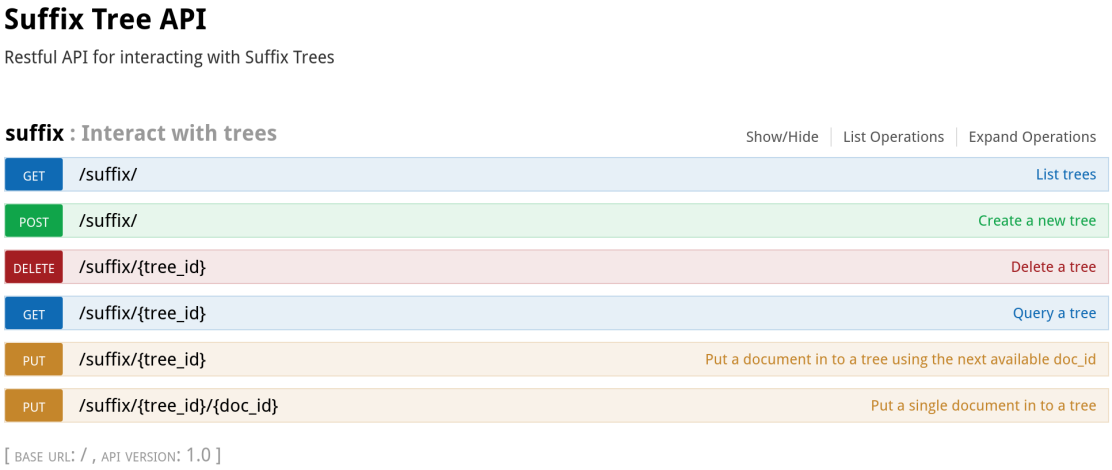



Figure 7: Swagger interface for Suffix Tree API

# 7 Conclusions

# 8 Critical Evaluation

## 8.1 Mistakes

### 8.1.1 Time-series Databases

Early on in the project, I had thought that it would be beneficial to store the raw data in a Time-Series based database such as InfluxDB, OpenTSDB or KairosDB. This would have allowed for the chaining together of events and in theory provided fast arbitrary access to data.

While this should have been the case, none of the databases mentioned were suitable. All of the open-source time-series databases seemed to be primarily focussed on the gathering of system metrics from a variety of sources and were not suitable for the large amounts of high frequency data points that came with the Seismic Data. The main sticking point with all three was that they were built for storing irregular data so each data point was stored with a timestamp meaning bytes per point and not bits.

None of my research was able to find a suitable time-series database and ultimately I settled on storing the observations in their raw format and then keeping meta data about them in an RDBMS.

### 8.1.2 Dataset

Initially I was provided with a large and mostly unsorted dataset (approximately 180GiB) spanning a years worth of observations from the Nabro Volcano in Eritrea. I boldly and subsequently fool-heartedly proceeded to attempt to analyse the dataset in full and with insufficient background knowledge to really understand what I was working with. This was also before I had realised the problems around normalising large datasets where most of the measurements were background noise.

Ultimately I was provided with a much smaller dataset of events to work from and this allowed my to get an actual prototype of the application running in its entirety.

## Bibliography

Beyreuther, M., R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann
2010. Obspy: A python toolbox for seismology.

Bormann, P.
2012. New manual of seismological observatory practice.

Harris, M., M. Levene, D. Zhang, and D. Levene
2016. The anatomy of a search and mining system for digital archives.

Haustant, A.
2016. Flask-RESTPlus fully featured framework for fast, easy and documented api development with flask.

Lin, J., E. Keogh, S. Lonardi, and B. Chiu
2003. A symbolic representation of time series, with implications for streaming algorithms. *Data Mining and Knowledge Discovery.*

Ukkone, E.
1995. On-line construction of suffix trees. *Algorithmica.*

Weiner, P.
1973. Linear pattern matching algorithms. *14th Annual IEEE Symposium on Switching and Automata Theory.*