GEOSPATIAL DATA STREAM PROCESSING IN PYTHON USING FOSS4G COMPONENTS

G. McFerren [a] *, T. van Zyl
[a] CSIR Meraka Institute, Meiring Naudé Road; Brummeria; Pretoria; South Africa - gmcferren@csir.co.za
[fb] School of Computer Science and Applied Mathematics, University of the Witwatersrand, 1 Jan Smuts Avenue, Braamfontein 2000, Johannesburg, South Africa - terence.vanzyl@wits.ac.za

KEY WORDS: Geospatial data streaming platform, Data velocity, Python, FOSS4G geospatial libraries, performance, SpS10-FOSS4G

ABSTRACT:

One viewpoint of current and future IT systems holds that there is an increase in the scale and velocity at which data are acquired and analysed from heterogeneous, dynamic sources. In the earth observation and geoinformatics domains, this process is driven by the increase in number and types of devices that report location and the proliferation of assorted sensors, from satellite constellations to oceanic buoy arrays. Much of these data will be encountered as self-contained messages on data streams - continuous, infinite flows of data. Spatial analytics over data streams concerns the search for spatial and spatio-temporal relationships within and amongst data "on the move". In spatial databases, queries can assess a store of data to unpack spatial relationships; this is not the case on streams, where spatial relationships need to be established with the incomplete data available. Methods for spatially-based indexing, filtering, joining and transforming of streaming data need to be established and implemented in software components. This article describes the usage patterns and performance metrics of a number of well known FOSS4G Python software libraries within the data stream processing paradigm. In particular, we consider the RTree library for spatial indexing, the Shapely library for geometric processing and transformation and the PyProj library for projection and geodesic calculations over streams of geospatial data. We introduce a message oriented Python-based geospatial data streaming framework called Swordfish, which provides data stream processing primitives, functions, transports and a common data model for describing messages, based on the Open Geospatial Consortium Observations and Measurements (O&M) and Unidata Common Data Model (CDM) standards. We illustrate how the geospatial software components are integrated with the Swordfish framework. Furthermore, we describe the tight temporal constraints under which geospatial functionality can be invoked when processing high velocity, potentially infinite geospatial data streams. The article discusses the performance of these libraries under simulated streaming loads (size, complexity and volume of messages) and how they can be deployed and utilised with Swordfish under real load scenarios, illustrated by a set of Vessel Automatic Identification System (AIS) use cases. We conclude that the described software libraries are able to perform adequately under geospatial data stream processing scenarios - many real application use cases will be handled sufficiently by the software.

1. INTRODUCTION

This paper concerns the description of a Python-based data streaming framework called Swordfish that is designed to be used in the transport and processing of streams of data that contain a geospatial or locational component.

We offer a brief introduction to the data streaming paradigm and provide some descriptive examples of data streaming software frameworks, before discussing the nature of geospatial data on streams. We then introduce the Swordfish framework – its architecture, approach to processing and implementation specifics – leading to a discussion on geospatial processing functionality and the Free and Open Source Software for Geospatial components that enable this functionality. Early performance insights are discussed. Finally, some usage scenarios are provided.

1.1 General Data Streaming Background

The concept of data streaming systems has long been recognised. In the (Babcock, et. al., 2002) synthesis and in (Lescovec et. al., 2014), a class of systems is identified that processes data arriving in "multiple, continuous, rapid, time-varying data streams" rather than data in sets of persistent relations. These data streams may be infinite or ephemeral and are often unpredictable (Kaisler, et.al., 2013).

The need for these kinds of systems results from the burgeoning of data arising from numerous sources including (Lescovec et. al., 2014), (Pokorný, 2006), (Kaisler, et.al., 2013), (Stonebraker, et. al., 2005):

- arrays of sensor networks or earth observing satellites continuously and variably transmitting multiple measurements of environmental parameters
- packets of data generated by network traffic
- social media
- science experiments and model outputs
- monitoring systems (cameras, electronic tolling)
- positions of moving objects (vehicles on roads, vessels at sea, parcels or cargo in delivery process)
- market trading systems, which can peak at several million messages per second, as illustrated by (FIF, 2013)

These data sources can produce very large volumes of data at rapid rates, in a variety of forms and complexities. It is difficult or infeasible to store all these data and analyse post-acquisition (Kaisler, et.al., 2013). Data streaming systems exist to process and extract value from such data as it is 'in-motion' with low latency.

* Corresponding author

Significant computational challenges arise as a result of these data stream characteristics, necessitating methods for ETL (extract, translate and load) of data, sampling strategies, aggregation and stream joining techniques, windowing approaches, stream indexing, anomaly detection, clustering, summarising of streams and many others as described by (Lescovec, et. al., 2014) and (Agarwal, 2007).

In (Stonebraker, et. al., 2005), it is argued that stream processing systems must exhibit eight properties:

1. Data should be kept moving – there should be no need to store data before processing and data should preferably be pushed rather than pulled to the processing components.
2. Support for a high-level processing language equipped with stream-oriented primitives and operators such as windows.
3. Resilience to imperfections of streams, such as partial data, out-of-sequence data, delayed or missing data and corrupted data.
4. Outputs should be predictable and repeatable (though as described above, techniques exist to sample and summarise streams of data, perhaps leading to a third quality statement around statistical significance).
5. Ability to store, access and modify state information and utilise such state in combination with live data, without compromising on low latency goals.
6. Mechanisms to support high availability and data integrity, for example through failover systems
7. Ability to scale or distribute processing across threads, processors and machines, preferably automatically and transparently.
8. Near instantaneous processing and response – provision of a highly optimised execution environment that minimises computations and communication overheads.

These properties provide guidance on the architecture and likely the goals of a data streaming system.

1.2 Geospatial Data Streaming Background

A significant amount of the data originating from the sources described previously, such as sensor networks, moving objects and social media has an explicit or implicit location or spatial context that can be utilised as data is processed.

This has some implications for data streaming software frameworks. Firstly, frameworks need to be capable of processing the extra volume of data necessary to describe location or spatial relationships. Second, it is important that data streaming components recognize geospatial data in the different forms it manifests in, so that the data can be accessed as efficiently as possible in pursuit of low latency. Thirdly, there needs to be a recognition that a significant number of the offline algorithms and processes that characterise geospatial computation (i.e. algorithms that have full knowledge of their input data) are not appropriate for the continuous, possibly infinite and often incomplete online nature of data streams, as noted by (Zhong, et. al., 2015). Algorithms and processes here need to deal with data as it arrives and may never have sight of the data again, since the complete data stream is unlikely to be captured in local computer memory.

This last issue hints at a need for a deeper discussion of classification of geospatial computation functions for streaming data. This is not dealt with here; for the purposes of this article it is enough to observe that different geospatial computations will be more adaptable to a streaming paradigm than others. This is driven by the complexity of the calculation and the amount of state or information completeness that is required by the calculation.

In concrete terms, a process that simply filters data by feature name or ID will be well suited to a streaming paradigm since it exhibits low complexity and no state requirement. A process to transform the spatial reference system of features is also easily fitted to a data stream, even though the process is more complex, since there is no state requirement to handle.

A process to join together two datasets based on a spatial relationship such as feature containment is more difficult or even intractable to implement in a streaming system. The state of both streams needs to be known, since each feature on one stream needs to be compared with every feature on the other stream; furthermore, the individual calculations could be expensive, depending on the complexity of the streamed features. This type of geospatial computation exemplifies the notion of an offline algorithm. However, a geospatial data streaming system arguably should offer this kind of functionality. Stream windowing functions like time-based windows (features for the last 10 minutes) or count-based windows (the last 100 features) offer a way to manage a limited amount of state. A spatial join could be performed on the features in small windows of the data streams, such that only features within the windows are compared to each other. This spatial join process also highlights the importance of spatial indexes on streams: in order to reduce latency and keep data moving, as per the eight properties of stream processing, a spatial index on the features in one window may help to reduce the number of containment calculations executed.

The geospatial stream processing approach may be deployed in answering a wide variety of geocomputation query types. Two classes of geospatial analysis are illustrative. (Xiong, et. al.,2004) provides some examples of queries that analyse the spatial relationships between features that change location over time:

- moving queries on stationary objects – petrol stations within a given distance of a moving car
- stationary queries on moving objects - counts of vessels inside a harbour, aeroplanes inside an airspace, cars on a road section
- moving queries on moving objects – the position of icebergs in relation to ship positions

(Zhong, et. al., 2015) demonstrate spatial statistical calculations over streams to generate spatial grids (for use in fire behaviour models) from point location data from sensor networks.

In broad terms, a geospatial data streaming framework should provide functionality for efficient structuring, filtering, aggregating, joining, transforming and analysing of the spatial component of data 'in motion'.

1.3 Data Streaming Implementations

A number of proprietary and open-source data streaming frameworks and query languages have existed in the last fifteen years. This paper does not intend to enumerate them, a task undertaken by (Jain et. al., 2008). Instead, we present here some modern, open source examples of data streaming frameworks that have influenced this work or are illustrative of the data stream processing domain. The frameworks briefly considered here are Storm, Samza, Kafka Streams and Spark Streaming.

In this viewpoint, we briefly describe, for each implementation:

1. Implementation origin – developers and driving use case of the implementation;
2. Data form – the atom of streaming, usually a tuple of values or a message consisting of key-value pairs;
3. Streaming transport – the underlying infrastructure used to move streaming data around;
4. Deployment and Execution infrastructure – the layers of software upon which framework resides and processing logic can be run;
5. Processing approach - batch/ mini-batch processing or per-atom processing;
6. Processing API – the kinds of functionality that are provided for processing data streams;
7. Domain-specific data model – the nature of a domain specific streaming data model , if present;
8. State Model and fault tolerance – many operations on streams require maintenance (and recovery) of state; streaming systems must be resilient to node failure and minimise or prevent message loss.

These tables should be viewed in terms of the eight stream processing system properties identified above.

Storm – http://storm.apache.org/	
Origin	Twitter Corporation – stream processing, continuous computation and distributed RPC.
Data form	Tuples. Represented as a named list of values of primitive or user-constructed types. The nature of the tuple is declared by the outputting node (a *Bolt* or *Spout*). Uses Kryo serialisation or Java serialisation internally. JSON is the interchange format when working in multiple languages
Transport	Storm has a message-oriented push-based approach, allowing it to abstract multiple message systems (such as AMQP, Twitter, Kestrel, JMS, Amazon Kinesis and Apache Kafka) and databases as *Spouts*.
Execution	Requires a Storm Cluster for managing (resource allocation, distribution and fault tolerance) computational topologies, using Zookeeper as the cluster manager; Java Virtual Machine with ability to write processes in multiple languages.
Processing	Per message processing – data are processed as received; functionality exists for batching, provided as a separate software layer
API	Computational topology/ graph oriented. An application is a topology deployed to a Storm Cluster. Object-Oriented. *Streams* are created from *Spouts* and are processed by *Bolts*, which are containers for arbitrary code. Windowing functionality can be added to *Bolts*. Also provides a basic SQL API. Provides a higher level API called Trident for micro-batching and harnessing the MapReduce style for gaining functionality for mapping, grouping, joining, aggregating and filtering data streams and persisting state in a number of databases/ caches
Data model	N/A – general purpose
State model and resilience	Local state storage relies on memory and HDFS. Trident allows persistence to external stores and provides an API for managing state and achieving fault tolerance.

Table 1: Streaming Frameworks - Storm

Samza - https://samza.apache.org/	
Origin	LinkedIn Corporation – used to process tracking and service log data and handle data ingest.
Data form	Kafka binary message format – header and variable length payload with gzip, snappy and lz4 compression. Serialisation format agnostic.
Transport	Uses Apache Kafka push-based messaging system, models message flow as a distributed commit log. Possible to use other transports; intention is Kafka, for its durability properties.
Execution	By default, Apache Hadoop YARN cluster manager for resource allocation, distribution and fault tolerance; Java Virtual Machine.
Processing	Per message processing – data are processed as received; functionality exists for batching but is not default.
API	Job-oriented MapReduce style API, Object-Oriented. *SamzaContainers* hold processing units called *StreamTasks* or *WindowableTasks* that process *Streams* (partitioned message streams).
Data model	N/A
State model and resilience	Local state storage per task into key-value database or transaction log, in memory or on disk. Allows a stream to be replayed, if necessary. Resilience achieved through cluster manager and underlying message transports.

Table 2: Streaming Frameworks – Samza

Kafka Streams - http://docs.confluent.io/2.1.0-alpha1/streams/index.html#kafka-streams	
Origin	Part of the Confluent platform for real-time streaming ETL
Data form	Data record in the form of a key-value pair
Transport	Apache Kafka push based messaging
Execution	Applications are built using the Kafka Streams Java library, but require the existence of a Kafka cluster of message brokers.
Processing	Per message processing – data are processed as received. Streams are represented as changelogs of a table and a table as a snapshot of a stream. Processing is partitioned on the topic of the data record, if necessary.
API	Computational Topology/ Graph oriented. A *Processor Topology* allows *Stream* and *Tables* to be processed by *Stream Processors*. There is a Domain Specific Language called Kafka Streams DSL that supplies these constructs and also facilities for windowing, joining, and aggregating streamed data
Data model	N/A – general purpose
State model and resilience	State can be stored in memory or in a process-local key-value datastore or other caches. Resilience cascades from the fault tolerance and scalability of the underlying Kafka software.

Table 3: Streaming Frameworks - Kafka Streams

Spark Streaming - http://spark.apache.org/streaming/	
Origin	University of California.
Data form	Spark Resilient Distributed Dataset (RDD) with Spark binary serialisation format or Kryo serialisation.
Transport	TCP sockets, files, Apache Kafka, ZeroMQ, MQTT, Amazon Kinesis, Twitter, Flume and Hadoop Distributed File System (HDFS) are the transports provided by Spark Streaming, but it is possible to use other transports.
Execution	Can run standalone on a Spark cluster or the Amazon Elastic Compute Cloud, but usually run in production using Apache Hadoop YARN or Mesos Hadoop cluster manager for resource allocation, distribution and fault tolerance; Java Virtual Machine, with wrappers for other languages.
Processing	Mini-batch processing – primarily data are read from a stream and batched for use by functions of the Spark Engine. Claims that this improves the ability of Spark Streaming to handle the imperfections of streams.
API	Provides a MapReduce style API, functional style. API provides a set of streaming related Transformations over D-Streams (Discrete Streams) including sliding windows, joins (stream-to-stream and stream-to-dataset), map, filter, reduce, union, count and transform (allowing any Spark function to be applied). Also provides an SQL and Dataframes API, which converts streams to tables and processes them using SQL operations
Data model	N/A – general purpose
State model and resilience	Local metadata and data state storage to memory by default and to HDFS if checkpointing is enabled. Allows a stream to be replayed, if necessary.

Table 4: Streaming Frameworks - Spark Streaming (Zaharia, et. al., 2012)

This short discussion of some of the features of various data streaming systems illustrates that there exist many approaches to constructing and deploying such a system, with varying levels of complexity and processing styles. It should be noted here that these ecosystems and frameworks primarily target the Java Virtual Machine.

1.4 Geospatial Data Streaming Implementations

Similarly to stream processing frameworks, there have been a number of implementations of geospatial data streaming frameworks over the last two decades. This section does not enumerate the various efforts, rather it highlights a few interesting exemplars.

PLACE (Mokbel, et. al., 2005) is one of the earliest implementations of a such a system. It was used to unearth and solve some of the fundamental issue of working with location in a streaming, specifically a continuous query context. PLACE introduced a number of pipelined spatio-temporal operators (e.g. a continuous query to ascertain whether a one feature was spatially inside another feature) and predicate-based windows (i.e. data only enters/ exits a query window if it satisfies/ no longer satisfies a predicate, such as a falling within a geographic area).

ESRI GeoEvent Extension for ArcGIS Server (ESRI, 2016) is ESRI's view on bringing streams of data to its large array of geospatial processing capability. This approach utilises ArcGIS Server and spatial analysis components of ESRI to act as a stream processing (described as an event processing) engine. Various streams of data such as sensor network output, social media feeds, etc. pass messages to this engine via an assortment of provided or custom developed *Input Connectors.* The messages get structured as *GeoEvents*, are acted upon and then streamed out via *Output Connectors.* This extension is aimed at spatial ETL, pushing of data to web applications, status updates in dashboard applications and real-time notification applications such as geofencing applications. The primary primitive supplied by this software is a *Filter*. Custom stream processors can be built to exploit the wide variety of processing capability available on the ESRI platform. ESRI provides what effectively amounts to a streaming data management platform, as it allows streams to be declared, controlled and accessed as a set of *Stream Layers, Containers* and *Services*.

IBM InfoSphere is used by (Zhong, et. al., 2015) as an infrastructure for supporting the deployment of a framework called RISER. RISER utilises stream processing for ETL of spatio-temporal data and as a spatial analysis engine performing spatial functions (such as interpolation) over sensor network data.

2. SWORDFISH SOFTWARE FRAMEWORK

2.1 Design Goals and Architecture

Swordfish is intended to provide a non-clustered stream processing software framework for the Python programming environment. Stream processing topologies, along which messages are passed, provide the main Swordfish structure. Nodes (processing units, sources and sinks) and edges (streams) can be distributed across machines, but do not have to be. The implication of a non-clustered architecture, e.g.. no default reliance on a Hadoop cluster, is that Swordfish stream processing topologies can be executed anywhere that Python can be installed; from a sensor gateway to a Desktop, from a single computer to a network of computers running in a cluster or cloud environment.

Python provides rich functionality for geospatial work, ranging from data translation libraries to machine learning and statistical analysis libraries. Furthermore, Python is a dynamically typed, general purpose programming language, providing great flexibility. Thus, Swordfish can utilise a functional style of programming, common to many of the streaming systems described, yet provide utilities from object-oriented software libraries. Swordfish processing topologies are dynamic, meaning that new nodes and edges can be established at runtime, rather than compiled into the topology.

The primary goal is to support the performance of spatio-temporal access, transformation and analysis against geospatial data streams from the kinds of systems illustrated previously, such as Automated Identification System (AIS) positional information from vessels, sensor networks monitoring phenomena like radiation levels, to monitoring networks for water and electricity usage, near real-time remote sensing data product feeds and social media feeds. Swordfish has been optimised in a number of places (data structures and streaming function primitives) to enhance performance, primarily by producing Cython code.

1.1.1 Common Data Model: Since Swordfish is primarily concerned with geospatial data, we have developed a data model based on a combination of ISO/ Open Geospatial Consortium (OGC) Observations and Measurements (O&M) (ISO, 2011), a conceptual schema principally for describing location aware sensor-based observations and the Unidata Common Data Model (CDM) (Unidata, 2014) that can be used and understood by all components of the framework. In general, Swordfish tries to translate data to this common data model as rapidly as possible after receiving it from a source, thereby enabling components to work seamlessly with the data, as soon as they get sight of it. Messages are moved through the system in a special high performance data structure, known as an AttributeDictionary, which aids processors in searching, indexing, extending and serialising the message payloads they receive. AttributeDictionaries are serialised to MsgPack structures by default when moved along streams and between processes. Other serialisation formats such as JSON and Google Protocol Buffers can also be used. Figure 1 shows a data item representing a single message, in JSON form, that would be passed through Swordfish:

```
"{'type': 'CF_SimpleObservation',
'phenomenonTime': '2015-09-
14T11:58:58.649575', 'result':
{'variables': {'pulse_value': {'units':
'litres', 'dimensions': ['int']}, 'time':
{'units': 'isoTime', 'dimensions':
['time']}}, 'data': {'pulse_value': [500,
500, 500, 0, 500], 'time': ['2015-09-
14T11:58:01.305564', '2015-09-
14T11:47:06.808586', '2015-09-
14T11:54:55.782008', '2015-09-
14T11:43:58.603956', '2015-09-
14T11:50:58.623827']}, 'dimensions':
{'time': 5}}, 'featureOfInterest':
{'geometry': {'type': 'Point',
'coordinates': [-25.753, 28.28]}, 'type':
'Feature', 'properties': {'id': 'Top
Reservoir'}}, 'observedProperty': {'type':
'TimeSeries'}, 'parameter': {'id': 49999},
'procedure': {'type': 'sensor', 'id':
'45030171', 'description': 'Sensus HRI-Mei
linked to meter 14787486'}}"
```

Figure 1: Swordfish Common Data Model example

This message is used in the testing process, representing the size and complexity of a typical message payload. Note the featureOfInterest property; it is in a structure known as GeoJSON (Butler, et. al., 2008), a de-facto community standard format that is well understood by numerous geospatial software packages.

2.1.1 Transport: To date, Swordfish is capable of read/write streaming of data over a wide and growing range of message transports, including Advanced Message Queueing Protocol (AMQP), ZeroMQ, MQTT, Redis, websockets and several in-memory structures. Adapters have been developed to harness social media streaming platforms like Twitter.

2.1.2 Execution: Swordfish has no requirements for a processing cluster to be present; it can run on a Desktop computer as part of a normal Python application. As such, it should be considered as a set of software libraries, implemented according to application needs. Swordfish can be executed in a distributed fashion using the Python code remoting platform called RpyC (RpyC, 2013), but this is not as transparently managed compared to the clustered systems. Inherently, as with most message passing systems such as Swordfish, a level of distribution is naturally possible through the use of message broker protocols that provide part of several of the transport implementations. By default, Swordfish uses in-memory transports, but in practice, data are usually received from transport mechanisms such as distributed message queues, e.g. MQTT. Software bindings/ adapters to such queuing systems need to be present for Swordfish to utilise them.

2.1.3 Processing: Swordfish is a message-oriented system with per-message processing semantics – data are processed as soon as received; no facility exists yet for batching of messages.

2.1.4 Application Programming Interface: Swordfish supplies stream processing utility via a set of primitives for describing nodes and edges in a stream topology and a set of primitives for adding actual processing functionality. Nodes are abstract *StreamProcessors* and would include *Sources* (e.g. a subscription to an MQTT topic, a file, a database), *Sinks* (places outside of the system where data can be passed to (e.g. database, web service endpoint, websocket, message broker), and concrete *StreamProcessors* (generic functionality executors). These nodes are connected by different types of *Streams*, which are components that abstract the underlying message transport protocol and provide a callback mechanism for 1...n *StreamProcessors* to receive messages off the stream, i.e. each *StreamProcessor* registers a callback with a *Stream*. *StreamProcessors* usually accept a function that will provide application logic. Swordfish implements optimised *StreamProcessors* that allow a MapReduce style of application composition: *Maps, Folds, Reduces, Joins, Filters*. *Maps* are generally used to transform or analyse each message and return an output (e.g. reproject the spatial data in each message). *Folds* are a specialised *Map* that allows a message to be compared to some representation of state that is passed in at the same time as the message , often the output of the previous message (e.g. a check to see if each message is further east in heading than the previous message). *Reduce* is a component that aggregates or summarises data and outputs a result, continuously or at certain time interval, count interval or other delta in the data (e.g. union the geometries of the last 100 messages). *Joins* allow one stream of data to be joined with another, following SQL style semantics of inner joins and left/ right outer joins (e.g. merging data from two streams based on feature ID or spatial location). *Filters* utilise some function to exclude data that does not meet some requirement from being output downstream (e.g. discard features that are not within a specific area-of-interest). A number of these operators will operate in sliding or tumbling fashion over a count or time window of the data on each stream; as described previously, it is nigh on intractable for a process such as a spatial join to maintain the state of all the messages it has ever had sight of. At the time of writing, Swordfish maintains state only via in-memory structures – no serialised state is managed.

2.2 Geospatial Functionality and Components

Geospatial utility is provided to Swordfish via a package of programming functions that can be invoked and passed through to Swordfish primitives like *Maps, Folds, Filters* etc. as arguments. These programming functions are provided by a set of well known Free and Open Source geospatial software libraries, wrapped with code to specialise them for use in the Swordfish streaming environment. All these programming functions understand and can transform, query and populate AttributeDictionaries of the common data model.

Swordfish spatial functionality is currently under active development. At the time of writing, Swordfish provides:

- Functionality for indexing spatial data and comparing messages to indexes (e.g. bounding box relationship tests an k-nearest tests) This functionality is provided by the Rtree (Rtree, 2016) Python wrapper of the libspatialindex software.
- Computational geometry functions for binary predicate tests, specifically whether feature geometries on streams cross, intersect or fall within other geometries (provided on the same stream, a different stream or via some other source). This functionality is provided via the Shapely (Shapely, 2016) Python wrapper of the GEOS computational geometry engine.
- Cartographic transformations (e.g. projections), provided by the Pyproj (Pyproj, 2016) wrapper of the Proj.4 library
- Forward, inverse and distance geodetic calculations (e.g. bearing, distance) also provided by Pyproj.

Some examples of how these functions could be deployed in Swordfish may be useful. We have deployed Swordfish to process AIS vessel data (hundreds of messages per second) and data from sensor networks monitoring water and electricity usage for large commercial sites, as well as radiation concentrations around industrial facilities. Specific examples include:

- Using Swordfish *Filters* with spatial indexes and binary predicate tests to ascertain whether or not a moving feature, such as a fishing vessel is present in an area-of-interest such as a Marine Protected Area.
- Using Swordfish *Fold* to split up a data stream of vessels from an AIS feed into individual streams of vessel positions, and performing a fix on the timestamp information in each positional message so that trajectories can be calculated
- Using Swordfish *Map* to transform Google Protocol Buffer structured data from radiation monitoring sensors into the common data model, so that interactive, real-time map visualisations could be created and interpolations performed.

2.3 Swordfish Performance

This section gives an quick indication of the kinds of throughputs that have been observed of Swordfish when processed on a 24 CPU computer with 32 gigabytes of RAM, running Ubuntu Linux version 14.04 and Python 2.7. Results are from tests that utilise a payload as described in the Common Data model section above. The message payload is a string approximately 1 Kb in size, represented as an AttributeDictionary, with each test repeated for 50 000 messages. In repeated tests, a portion of which are illustrated in Table 5, Swordfish demonstrates some performance characteristics as follows:

Test	Throughput (average messages per second)
Raw throughput of messages via a StreamProcessor	> 660 000
In-memory streams	> 300 000
Simple Filter, in-memory stream	~ 240 000
Simple Map, in-memory stream	~ 66 000
Simple Fold, in-memory stream	~ 16 500
Simple Reduce, in-memory stream	~ 13 000
Simple Join, in-memory stream	~ 11 000
Distributed Map, 6 processes , in-memory streams, elapsed time ~ 10 seconds)	~ 6 350 per process ~ 30 000 for whole batch
Rtree bounding-box intersection test *Filter,* in-memory stream	~ 7000
Format AIS messages to Common Data Model, Swordfish *Map*, MQTT stream transport	> 5000

Table 5: Indicative performance results

These are early results, but show that Swordfish can stream and process high velocity data streams. The payload here is quite large; numbers improve drastically for a small, non-spatial, no-common data model message (e.g. a key-value pair of an integer and a short string). Streaming systems often claim throughputs of > 1 000 000 messages per second, but we feel the numbers we illustrate are more likely to be found in practice when dealing with geospatial data streams. It is notable that throughput slows when geospatial functionality is applied; these results nevertheless show Swordfish as capable of throughput of an order of magnitude greater than our highest velocity streams (merged Satellite and Terrestrial AIS receivers). The distributed/ multiprocessing capability of Swordfish may reduce any throughput bottlenecks when it is necessary to scale the system.

3. SUMMARY AND FURTHER WORKS

In this article we discuss geospatial data stream processing and introduce the Swordfish stream processing framework, highlighting some of its spatial capabilities. We indicate that Swordfish offers sufficient throughput capability to allow application developers using Python to build online geospatial systems for a number of potential use cases. A significant effort is needed to expand the geospatial functionality (particularly to move beyond computational geometry and indexing functionality) and perhaps optimise it as necessary. Effort needs to be undertaken to ensure that Swordfish is stable for long running applications, though its early deployment in particular use cases suggest it is reasonably stable. Swordfish is currently limited to holding state in memory; further work may be necessary to develop mechanisms to serialise state, especially in use cases where recovery of the streaming topology state may be necessary. A long term view of Swordfish development is the provision of a streaming data management platform, (as is provided by ESRI GeoEvent Extension, and the Confluent platform). We are investigating the process for open-sourcing Swordfish; organisational policies enforce a technology evaluation process before open-source licenses can be applied to software and code placed under an open-source management model.

REFERENCES

Aggarwal, C.C.(ed.), 2007. *Data streams: models and algorithms* (Vol. 31). Springer Science & Business Media LLC, 233 Spring Street, New York, NY 10013, USA.

Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J., 2002. Models and Issues in Data Stream Systems. ACM PODS, June 3-6 Madison, Wisconsin, USA. pp 1-16 .

Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T. and Schmidt, C. 2008. The GeoJSON Format Specification. http://geojson.org/geojson-spec.html (04 Mar. 2016)

ESRI, 2016. ArcGIS GeoEvent Extension for Server. http://www.esri.com/software/arcgis/arcgisserver/extensions/geoevent-extension . (04 Mar. 2016)

FIF, 2013. Financial Information Forum Market Data Capacity Charts June 2013 Data, 2013. Financial Information Forum. https://fif.com/docs/2013_6_fifmd_capacity_stats.pdf (04 Mar. 2016)

ISO 19156:2011, Geographic information -- Observations and measurements, 2011.
https://www.iso.org/obp/ui/#iso:std:iso:19156:ed-1:v1:en (04 Mar. 2016)

Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H, Cetintemel, U., Cherniak, M. , Tibbetts, R. and Zdonik, A., 2008. Towards a Streaming SQL Standard. PLVLDB 08, August 23-28, Auckland, New Zealand. pp. 1379-1390

Kaisler, S., Armour, F., Espinosa, J.A. and Money, W., 2013, January. Big data: Issues and challenges moving forward. In System Sciences (HICSS), 2013 46th Hawaii International Conference on (pp. 995-1004). IEEE.

Leskovec, J., Rajaraman, A., Ullman, J.D., 2014. *Mining of Massive Datasets* 2nd Edition, Cambridge University Press, New York. ISBN: 978-1-107-07723-2 pp 131-162

Mokbel, M.F., Xiong, X., Aref, W.C. and Hammad, M.A., 2005. Continuous Query Processing of Spatio-Temporal Data Streams in PLACE. *GeoInformatica* 9:4, Springer Science & Business Media, Inc. pp. 343–365

Pokorný, J., 2006. Database architectures: Current trends and their relationships to environmental data management, *Environmental Modelling & Software*, 21(11). pp. 1579-1586.

Pyproj, 2016. Python interface to PROJ4 library for cartographic transformations. http://jswhit.github.io/pyproj/ . (04 Mar. 2016)

RpyC, 2013. RpyC, Remote Python Call.
https://rpyc.readthedocs.org/en/latest/ . (04 Mar. 2016)

Rtree, 2016. Rtree: Spatial indexing for Python.
http://toblerity.org/rtree/ . (04 Mar. 2016)

Shapely, 2016. Shapely. http://toblerity.org/shapely/ . (04 Mar. 2016)

Stonebraker, M., Çetintemel, U., and Zdonik, S. 2005. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Record* 34. pp. 42-47

Unidata, 2014: Unidata's Common Data Model Version 4. Boulder, CO: UCAR/Unidata Program Center. http://www.unidata.ucar.edu/software/thredds/current/netcdf-java/CDM/ . (04 Mar. 2016)

Xiong, X., Mokbel, M.F., Aref, W.c., Hambrusch, S.E., and Prabhakar, S., 2004. Scalable spatio-temporal continuous query processing for location-aware services. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management. Santorini Island, Greece. pp. 317-326

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. and Stoica, I., 2012. Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing. Technical Report No. UCB/EECS-2012-259, Electrical Engineering and Computer Sciences University of California at Berkeley, USA. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.pdf

Zhong, X., Kealy, A., Sharon, G. and Duckham, M., 2015. Spatial Interpolation of Streaming Geosensor Network Data in the RISER System. In: *Proceedings of* the *Web and Wireless Geographical Information Systems: 14th International Symposium, W2GIS* , Grenoble, France. pp. 161-177