Querying XML Data Streams from Wireless Sensor Networks: An Evaluation of Query Engines

Martin F. O'Connor*, Kenneth Conroy*†, Mark Roantree*, Alan F. Smeaton† and Niall M. Moyna†‡
*Interoperable Systems Group, Dublin City University
†CLARITY: Centre for Sensor Web Technologies, Dublin City University
‡School of Health and Human Performance, Dublin City University
{moconnor,kconroy,mark.roantree}@computing.dcu.ie, {alan.smeaton,niall.moyna}@dcu.ie

***Abstract*—As the deployment of wireless sensor networks increase and their application domain widens, the opportunity for effective use of XML filtering and streaming query engines is ever more present. XML filtering engines aim to provide efficient real-time querying of streaming XML encoded data. This paper provides a detailed analysis of several such engines, focusing on the technology involved, their capabilities, their support for XPath and their performance. Our experimental evaluation identifies which filtering engine is best suited to process a given query based on its properties. Such metrics are important in establishing the best approach to filtering XML streams on-the-fly.**

***Index Terms*—XML, Streaming, Query, Filtering Engine, Sensor.**

I. Introduction

Sensor devices provide a bridge between the physical world and the digital domain. They facilitate an automated quantification and analysis of real-world events and optional automated responses should they be required. Sensors were traditionally deployed in domains such as environmental monitoring, home automation and traffic control. The recent advances and availability of wireless networks has increased the rate of adoption of ubiquitous wireless sensor devices. In particular, the healthcare and sport science domains have been quick to exploit the new functionality and flexibility afforded by these devices.

The proliferation of wireless sensor networks present new and unique challenges for the management and querying of the data streams they generate. Traditionally, data streams were employed in activities such as the dissemination of news feeds and stock market updates. The primary purpose of data stream queries were to monitor key values and flag them should they move outside a user-defined boundary. XML, a key format for system interoperability, has become the *de facto* representation for the encoding and transmission of data streams. Each new sensor device leads to the provision of increased and more complex functionality. As a result the wireless sensor networks in which they participate necessitate a query filtering engine at the data management layer to facilitate the efficient and effective querying of these data streams.

A. Background

An area of active research is the deployment of wireless sensor devices to monitor physiological and movement characteristics of athletes. The use case presented in this paper is contextually based on the integration of sensor streams in personal health networks [1]. The study was a collaboration between sports scientists and data engineers whereby a series of experiments were performed on teams playing Gaelic (Irish) football. Each player wore several biometric sensors monitoring and recording a number of physiological responses. The data was later uploaded from the sensor devices through USB connections to a computer in order to be queried. Indeed the focus of this earlier study lay in the provision of semantic enrichment and data integration services as part of a wider data infrastructure and query management system for a personal health sensor network environment.

Our study is a collaboration between sports scientists and data engineers part of which involved three match officials at a 2008 inter-county Gaelic Football Championship match were equipped with a prototype *wear and forget* textile-based physiological wireless sensor vest. The textile-based wireless sensor records the physiological responses of the wearer by integrating electrodes into the fabric using a liquid crystal polymer film. The data recorded includes heart rate and respiration rate, measured and broadcast to a base station PC wirelessly in real-time. GPS coordinates of the wearer were integrated with the device's data at a later time. In tandem with the wireless broadcasting of the sensor devices (vest and GPS), three live video feeds monitoring each of the match officials throughout the game was also captured, which was subsequently manually annotated as to the activity of each match official, which provided another data stream for encoding in XML.

B. Motivation

The health and human performance specialist requires the ability to query data in real time, during the sports event as it is being played, in order to prescribe actions like rehydration, or substitution. The wireless sensor vest prototype enables real-time monitoring and facilitates automatic triggering of notifications when an athlete's physiological readings are above or below a predefined threshold. These triggers may be simple (e.g. when heart rate reaches a certain value) or complex (two or more biometric readings satisfying combined conditions). The coach may view the video and data corresponding to the triggered event in order to determine if the biometric reading

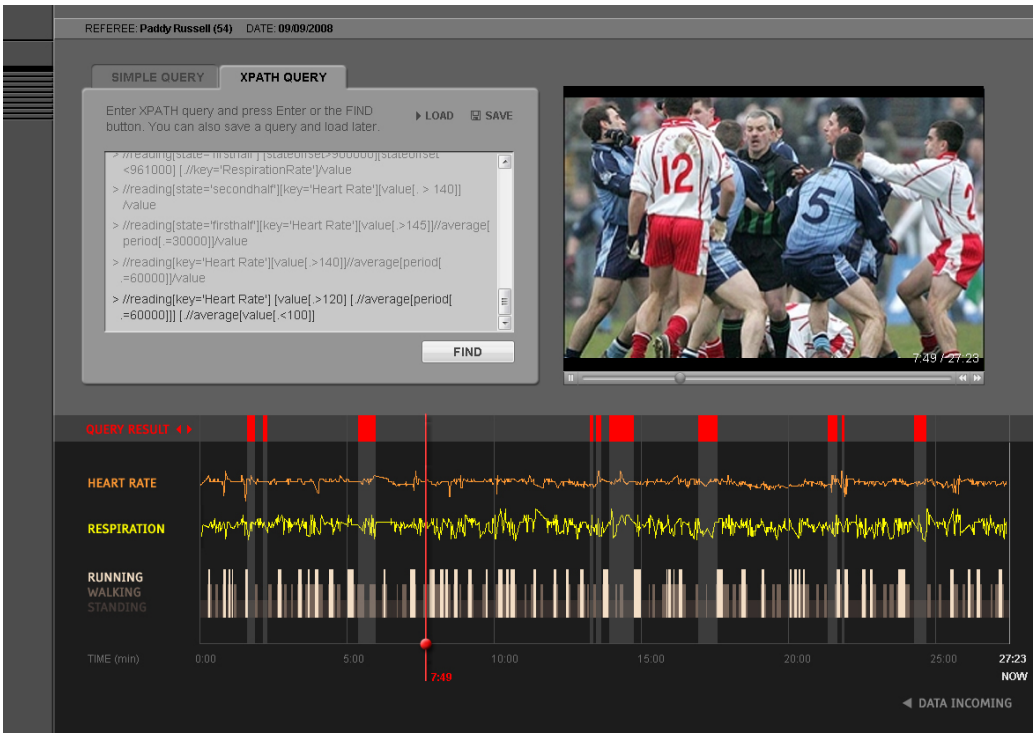


Fig. 1. Real Time Analysis of an Athlete during a Football Game

was appropriate to the action being performed by the athlete or wearer. This enables the coach to take corrective action such as substituting the player during a game, or changing the drill in the current training session. These triggers may be implemented as queries on the streaming data. The coach can effectively design individual training programs and tweak their implementation through real-time analysis of their biometric data so as to optimise the performance of each athlete. The streaming biometric data and captured streaming video of the football game are illustrated in Figure 1 along with sample queries to isolate and highlight significant fluctuations in average heart rate values.

This paper addresses the need for an efficient XML filtering engine, analysing several approaches with a particular focus on the capabilities and performance of such systems. Our motivation is to evaluate existing tools and technologies so as to be able to invoke the right query engine for a given arbitrary query. The ability to provide a filtering agent at runtime to dynamically select the most appropriate and efficient streaming query engine to process any given query based on its properties is the principle motivation of this work.

C. Contribution

In this paper, we provide a detailed analysis of five existing streaming query engines. The engines evaluated are YFilter, SPEX, XSQ, XMLTK and XPA. This analysis covers the functionality of each engine including their support (or lack thereof) for the XPath language, including wildcards, multiple predicates and complex queries. A particular focus is placed on YFilter and SPEX which are subject to extensive experimental analysis. We analyse a number of different queries to find out which query engine is best suited to process an arbitrary query according to its properties. We identify the strengths and weaknesses of each streaming query engine and present our recommendations for which to use based on our analysis.

The structure of this paper is as follows. §II presents a detailed overview of the five streaming query engines, outlining their functionality and detailing how they are implemented. §III presents a comprehensive comparison of their features, capabilities and the respective advantages and limitations of each approach. §IV provides a description of the experiments performed using the SPEX and YFilter engines and detailed analysis of the results. In §V we present our conclusions.

II. Related Research in Query Engines

In this section we provide an overview of each of the five XML filtering engines examined and outline their implementation details. Queries of XML data streams are specified in the XPath [2] or XQuery language [3] (of which XPath is a subset). An XPath expression is a sequence of location steps; each comprising of an axis, a node-test and optional predicates, that can traverse and extract data from the XML tree. XPath can be expressed in its full, expanded syntax or in a more compact abbreviated form. Each XML query engine supports varying subsets of the XPath language and will be detailed below. The XML data arrives to the system in stream form, which may be readily processed by the Simple API for XML (SAX) [4] parser. An overview of each of the five streaming query engines now follows.

A. SPEX

The SPEX system [5] for evaluating XPath queries against XML data streams is built upon frameworks for removing reverse axes (such as ancestor and preceding) from XPath queries and replacing them with the equivalent forward axes. Evaluation is achieved using a network of *pushdown transducers*. A pushdown transducer is a linear bounded automaton with outputs. The network consists of a tuple detailing the state of the pushdown transducer(s), the input and output alphabet, the stack alphabet and a transition function. The transducers act as nodes on a directed acyclic graph known as a transducer network. The pushdown transducers use their stacks to track the depth of the tree being examined. The processing of an XML stream is effectively a depth-first traversal of the tree in order to evaluate queries [6].

The SPEX Query processor comprises four steps. The first step requires rewriting the XPath query to an equivalent query consisting of forward axes only, eliminating any reverse axes. This is a necessary feature of processing streaming data due to its dynamic nature: one cannot go back through data paths. This step also includes optimisations focusing on pruning redundant computations to improve the evaluation of forward XPath queries. The second step compiles the forward axes-based XPath query into a logical query plan. Compile-time optimisations (where applicable) also occur at this stage. The third step computes a physical query plan from the logical plan. The fourth and final step processes the physical query plan. This step essentially performs a depth-first traversal of the XML tree representing the stream. The transducer stacks are employed to track the depth of the processed nodes. This allows the computation of forward XPath axes in a single pass.

Queries are received by SPEX in expanded XPath format and the answers computed are buffered because predicate evaluation may occur later downstream and prune the final result set. Structural filters (another type of transducer) are employed to minimise the stream traffic between transducers in the network [5].

B. YFilter

YFilter is a non-deterministic finite-state automaton (NFA) based approach to XML stream processing. The NFA consists of an alphabet, a set of states including a start and final state and transition relations. YFilter seeks to provide an efficient filtering engine for thousands of query specifications [7] [8]. The NFA based representation of XPath expressions is the key innovation and permits all queries to be combined into a single machine, exploiting commonality among queries. Further advantages of this approach are a small number of machine states and the support for incremental machine construction [7]. To deal with nested paths (e.g. a predicate which includes other paths) YFilter uses a query decomposition scheme which takes advantage of the shared-path processing. The query evaluation results are returned by a special post-processing technique. It should be noted that all XPath expressions can be converted into a regular expression and thus, there exists an NFA to accept any XPath query.

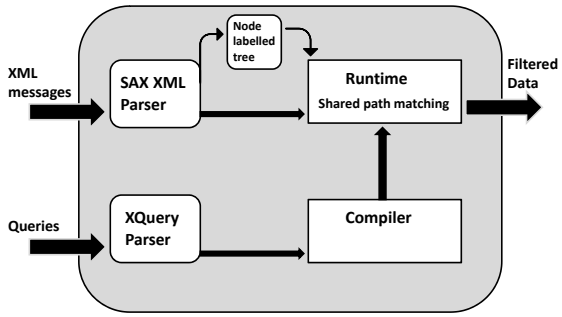


Fig. 2. YFilter Processing Architecture

The YFilter processor proceeds as follows: The NFA's are combined to form a single NFA with one initial state. From this state (and all subsequent states) transitions are made until an accepting state is reached or a state is reached which does not match the corresponding state in the query. YFilter uses a separate selection operator for predicate evaluation. The two approaches *Selection Postponed* and *Inline*, are defined as follows: *Inline* evaluates predicates as soon as the addressed elements are read from the stream; *Selection Postponed* delays predicate evaluation until the corresponding XPath expression has been entirely matched. The YFilter Processing Architecture is illustrated in Figure 2.

YFilter performs an order of magnitude faster than its predecessor XFilter which employed an Finite State Machine (FSM) based approach. The NFA approach of YFilter is efficient for the non-deterministic operators which form an important part of XPath [7].

C. XSQ

XSQ presents an XML filtering engine based on the hierarchical arrangement of pushdown transducers augmented with buffers [9]. It is memory efficient and provides high throughput. XSQ implements all aspects of XPath 1.0, excluding the reverse axes and the *position()* functions.

The pushdown transducers (PDT) are pushdown automata with actions defined along the transition arcs of the automaton [9]. At each step, the state transition is determined by the input symbol and the symbols on the stack. Each SAX event makes a state transition based on the transition table. The PDTs do not have a buffer and thus cannot answer XPath queries with predicates. To overcome this limitation, the Buffered Pushdown Transducer (BPDT) extends the PDT with a buffer, organised as a queue with several buffer operators. Each BPDT may encode a single location step and thus all the BPDTs are combined into one Hierarchical PDT (HPDT) in order to process entire XPath queries. The position of the BPDT in the HPDT encodes the results of all predicates. The streaming input to the HPDT comes from the SAX parser [9].

On receipt of an XPath query, it is first parsed by the XPath parser into a sequence of location steps, each step consisting of an axis, a node-test and an optional predicate. The object of a predicate falls into one of five categories or templates defined by XSQ. These templates provide the basis

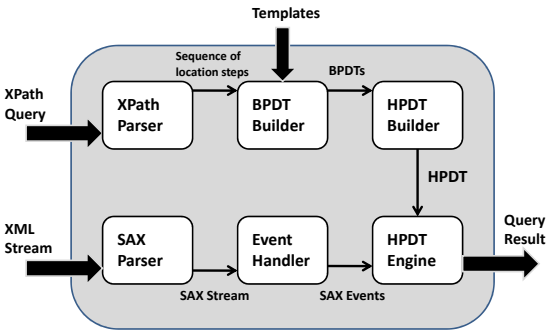


Fig. 3. XSQ Processing Architecture

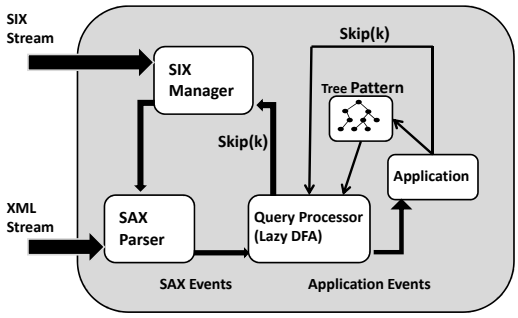


Fig. 4. XMLTK Processing Architecture

for building a BPDT for each location step. The set of BPDTs is stored indexed by their source states. The XSQ Processing Architecture is illustrated in Figure 3.

The HPDT builder connects the BPDTs into one by assigning a unique state ID to each state in all BPDTs, which is maintained in the transition arcs of the HPDT. All states are stored in a single set following the assignment. At runtime the HPDT engine must execute the HPDT produced by the builder. It must maintain the stacks, buffers and other runtime objects. The SAX parser calls a user-defined event handler to process events. The event handler records the depth of the event and ensures the input is well formed XML [10].

D. XMLTK

The XML Toolkit (XMLTK) consists of two core components. The first component consists of a collection of standalone tools such as sorting and aggregation utilities which can be combined to facilitate a more complex processing architecture. The second component is an XML processor for XML streams with an emphasis on scalability [11]. Together these components make up a framework for lightweight and high-performance XML stream data processing.

A core technology introduced by XMLTK is SIX, a Stream IndeX designed for stream processing. The SIX of an XML stream is a sequence of byte offsets in the stream that the XPath processor recognises so as to skip sections of the data stream. A SIX is computed once for each packet and must be routed together with the packet. The size of the SIX is kept small in order to have a nominal effect on throughput and memory usage. The XMLTK toolkit provides functionality to XML streams similar to that of UNIX commands on text files such as `grep`. When used in conjunction with XPath these tools can allow the user to perform complex operations on XML streams and are designed to scale. In order to further improve performance, the developers provided a binary format for XML and SIX, replacing tags and attributes with integers tokens and allowing recognition of some atomic data types. They also defined their own tokenised SAX (TSAX) parser. The TSAX parser provides a single interface to both standard and binary XML.

The XMLTK processor consists of a stream API extending the tokenised SAX parsing model and is illustrated in Figure 4. A query is presented as a tree. The tree nodes are labelled with variables and the edges labelled with XPath expressions. The XPath processor exploits these labels to detect when a match with an input XML stream occurs. However, the modification of the SAX parser comes with a performance penalty because it necessitates an extra hash table lookup.

The processor converts the query tree into a non deterministic finite-state automaton (NFA), from which the deterministic finite automaton (DFA) is computed. The processor is guided by the TSAX events. The stack may only grow to a height equivalent to the maximum depth of the XML stream [11]. This application achieves constant throughput independent of the number of XPath expressions. In order to prevent the exponential increase in the number of states in the DFA compared to the NFA, the DFA is constructed lazily.

E. XPA

XPA [12] presents a SAX based approach for XPath query evaluation. The input query is translated into an automaton consisting of four types of transitions. This number of automata requires a small memory footprint supporting the fast evaluation of the input XML stream. XPA can support XPath query evaluation on infinite XML data streams and each SAX event is read once. However, this requires the stream be parsed in a single pass and consequently all backward axes are rewritten into equivalent forward axes. The advantages of this design are tempered by the potential for very large queries resulting from the rewrite.

The XPA processor proceeds as follows: The SAX input stream is first converted into a binary SAX event stream, supporting *child, sibling* and *parent* events and *self* node tests. This is performed in two steps. Each SAX event character and attribute value pair are transformed into binary SAX event sequences. In the second step, the start and end elements are replaced with events according to the set of rules defined in [12]. The binary SAX events are used as input symbols for the stack of XPath automaton, constructed for the XPath query. The queries after decomposition and normalisation contain only three types of axes. These are in turn converted into XPath automata for which a stack of active states is maintained. The input SAX event is converted to a binary SAX event stream and acts as input for the XPath automata. Each XPath query is decomposed into a set of filter-free path queries.

	Xpath Query representation	Results representation	Xpath Axes supported	Wildcards	Filter Predicates	Text data	Attributes	Position Functions	Multiple Predicates	Comparitive Ops	Conjunctive (and)	Ranges
Yfilter	Abbreviated	relevant results	**child, descendant-or-self.** No sibling axes support	✓	✓	✓	✓	✓	✓	Only on position()	NO	NO
SPEX	Expanded	relevant results	**child, descendant-or-self.** No sibling axes support	✓	✓	✓	NO	NO	✓	!=. = only	✓	NO
XSQ	Abbreviated	relevant results	**child, descendant-or-self.** No sibling, reverse axes support.		✓	✓	✓	NO	✓*	✓		
XMLTK	Abbreviated	relevant results	**child, descendant-or-self.** No sibling axes support	✓	✓	✓	✓	✓	✓*	NO	NO	NO
XPA	Expanded	relevant results	**All Axes supported**	✓	✓	✓	✓		✓			

***conditions apply**

Fig. 5. Evaluation Metrics and Results for the Five Streaming Query Engines

The XPath query is then normalised by rewriting each path expression such that they contain only the location steps *firstchild, nextsibling* and *self*. The resulting filter-free XPath query is used to create the XPath automaton. For each location step in a query path, an atomic XPath automaton is computed. Finally they are concatenated to form the complete XPath automaton [12].

III. Evaluating Query Engines for Sensor Streams

The query engines described in §II will be used to query, in real time, sensor data generated during experiments involving physical exercise during championship football matches, as well as laboratory-based training. Thus, it is necessary to evaluate how powerful their query features are, and those engines that are best suited to performing different types of queries. This section provides an detailed analysis of the functionality of each of the five stream query engines based on the particular set of user requirements our application demands. Each streaming query engine supports a subset of XPath to various degrees. This section qualifies those differences and summarises the benefits and limitations of each engine.

A. Evaluation Template

The metrics chosen as part of our evaluation of the XML streaming engines are representative of the requirements of the domain specialist, and the diversity and key properties of an XPath query. The metrics by which the query engines were evaluated and compared are as follows:

- The representation format in which the XPath query must be received by the query engine.
- The relevance of the results returned to the query given.
- The XPath axes supported by the streaming query engine.
- Support for wildcards.
- Support for filter predicates (e.g. test a node for a condition).
- Support for the XPath *text()* function.
- Support for attributes.
- Support for the XPath *position()* function.
- Support for multiple predicates.
- Support for the comparative operators.
- Support for the conjunctive *and* operator.
- Support for range queries.

B. Evaluation Overview

Figure 5 displays the results obtained from researching the XML filtering engines. The greyed-out boxes represent information that could not be determined. Both SPEX and XPA require the XPath queries to be represented in expanded form. The remaining three engines accept the abbreviated form of XPath. All of the engines provide axis support for child and descendant-or-self axes. XPA is unique in that it supports all XPath axes. Sibling axes are not supported by YFilter, SPEX, XSQ or XMLTK. Furthermore, XSQ lacks support for reverse axes and *position()* functions.

C. Analysis

1) SPEX:: SPEX primary limitation is the lack of support for attributes or the *position()* functions in XPath. SPEX does have support for comparative operators on text data, however this support is restricted to the *equals* and *not equals* operators. The remaining operators, such as *greater-than* and *less-than* cannot be applied to text data. Consequently, range queries cannot be specified in SPEX. Nevertheless, the Boolean operators *and* and *or* are available to specify predicates in SPEX.

2) YFilter:: YFilter has limited support for XPath expressions, namely the child and descendant-or-self axes. In addition to the details presented in Figure 5, YFilter can also process queries with nested paths. All path queries are combined to form the NFA where common prefixes are represented. The need to support multiple transitions could cause performance problems which YFilter avoids by transforming the NFA into a DFA. In prior evaluations of the system, it was shown that these performance concerns are overstated, with flexibility and ease of maintenance proving more advantageous

Query	Abbreviated Syntax
1	//measurement/reading[state]/value
2	//measurement[reading]/*/value
3	/healthSense/sensorData/sections/section/measurement[reading]/reading/value
4	//reading[.//key[text()="Respiration Rate"]]/value
5	//reading[.//key[text()="Respiration Rate"]][.//state[text()="firsthalf"]]/value
6	//measurement[@state="firsthalf"]//reading[.//key[text()="Respiration Rate"]]/value
7	//measurement[reading]/reading[state]/value
8	/healthSense/sensorData/sections/section/measurement/reading[state]/value
9	/healthSense/sensorData/sections/section/measurement/reading[state]/averages/average/value
10	//reading[.//key[text() ="Heart Rate"]]//average[time[text()=60000]]/value
11	//reading[.//key[text() ="Respiration Rate"]]//average[time[text()=30000]]/value
12	//measurement[position()=520]/ reading[.//key[text()="Respiration Rate"]]/value
13	//measurement[position()=245]/*/value
14	//measurement[position()=245]/reading[.//key[text()="Heart Rate"]]/value
15	//reading[.//key[text() ="Respiration Rate"]]/averages/average/value
16	//measurement[@time=1220836800000]/reading[./key[text()="Heart Rate"]]/value
17	//measurement[position()<10]/reading[.//key[text()="Heart Rate"]]/value

Fig. 6. List of Input Queries

Query	Expanded Syntax
5A	/desc::reading[child::key/child::text() = 'Respiration Rate'][child::state/child::text() ='firsthalf']/child::value
5B	/desc::reading[child::key/child::text() = 'Respiration Rate' and child::state/child::text() ='firsthalf']/child::value
10	/desc::reading[child::key/child::text() = 'Heart Rate']/desc::average[child::time/child::text() = '60000']/child::value

Fig. 7. Sample Queries in Expanded Form

than faster path processing [7], [13]. Of the two approaches available for predicate evaluation, *Selection Postponed* results in better performance in comparison to *Inline* because of its ability to prune the potential set of queries. Further advantages of YFilter are its ability to support recursive documents, support queries with multiple wildcards and the relatively small number of machine states required to represent the large numbers of XPath expressions.

3) XSQ:: Previous experimental evaluation of XSQ focused on throughput, memory usage and features supported [10]. The experiments involved the use of non-streaming engines such as Saxon, which consumed much more memory than the streaming systems. For the engines which do not support predicates the size of the input data has a nominal effect on memory requirements. In order to support predicates, buffering is required and thus XSQ may require a large amount of memory depending on the size of the data stream and the number of queries to be processed [9]. Although XSQ supports multiple predicates, this support is subject to a number of conditions, namely each node may have at most one predicate and each predicate can contain path-to-value comparisons with a path size of one (containing the axis: child, text or attribute) [12] [10].

4) XMLTK:: XMLTK support for predicates is weak. There are restrictive conditions governing the use of predicates, such as disallowing the *position()* function to appear after a descendant-or-self axis and position predicates may not follow an other predicate. XMLTK offers additional UNIX-like functionality such as simple tools for complex transformations. XMLTK is designed to scale well and allow transformations on very large XML streams. The tokenised SAX (TSAX) events defined for the XML parser reduces the size of the data by a factor of two, and the result is two fold improvement in speed. The authors present a suite of tests conducted on XMLTK and demonstrate that for large numbers of expressions, throughput using this technique is several thousand times faster than XFilter, the predecessor to YFilter. Their evaluation does not include the techniques discussed in this paper. The high throughput achieved comes at the cost of greater memory requirements for storing the DFA rather than NFA [11]. Nevertheless, the lack of support for predicates is a considerable limitation which in a real-world scenario effectively outweighs XMLTK benefits.

5) XPA:: XPA provides for an XML Filtering Engine which theoretically appears superior to the other engines presented here. XPA fully supports XPath 1.0 including sibling axes, and is the only streaming query engine to do so. The implementation has not yet been released so further analysis of the system performance is not possible. Previous evaluation analysis suggest XPA is an efficient system offering the greatest range of functionality [12]. Query test suites were performed by the developers on both XPA and YFilter. The results show a substantial performance improvement for XPA over YFilter, with XPA requiring far less memory. YFilter consumed double the memory for document storage compared to XPA, and was prone to out of memory exceptions when dealing with very large XML documents. The throughput rate for XPA was measured as 40MB/s, an exceptionally high result. XPA had a

better evaluation time with respect to the size, especially when dealing with documents larger than 50MBs. The exponentially larger queries resulting from rewriting backward axes results in a slight decrease in speed [12]. XPA is an efficient filtering engine, particularly suited to very large XML documents and streams. However, as the implementation is not yet available, it has been excluded from the experiments presented in this paper.

IV. Experimental Evaluation

In this section we examine the performance of two streaming query engines SPEX and YFilter. Recall that our primary motivation is to examine the performance of each of these engines using a comprehensive set of queries in order to identify the suitability of each engine to process a given query according to its properties or classification. A query may be classified according to its diversity and combined usage of axes, syntactic constructs, operators, functions and various other features as illustrated in Figure 5.

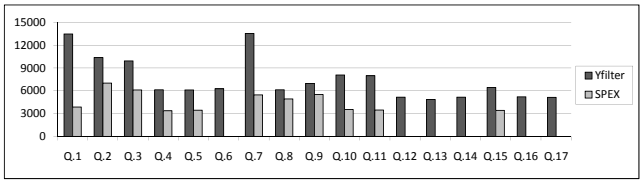
A. Experimental Setup

The experimental setup consisted of a prototype vest worn by three match officials while participating in a GAA Championship football match. The prototype *wear and forget* textile-based physiological wireless sensor vest, manufactured by Foster-Miller, records the physiological condition of the officials by integrating electrodes to monitor Heart Rate with a breathing sensor. The electrics and sensors are combined with a liquid crystal polymer (LCP) film. The data recorded includes heart rate and respiration rate measured and broadcast wirelessly in real-time. The data was collected in real time, and the resulting XML files make up our dataset and were used as the input files to the Query Engines we examined.

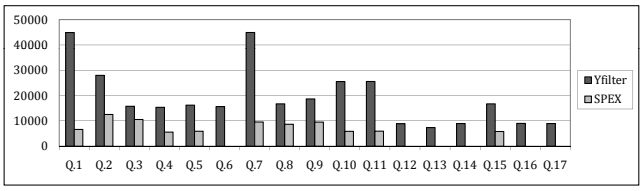
All of the experiments reported here were performed on an Intel Celeron 1.4GHz laptop with 512MB RAM running Windows XP Pro with Java JVM 1.6.0_06. The Java maximum heap size was set to 128MB. Both Spex and YFilter streaming query engines provide a Java implementation and we augmented our experimental setup with our own simple Java application to invoke our experiments and record the results. Each query was run four times and the times recorded. The first run was treated as a cold run and thus ignored. The remaining three times were averaged and this average is the time recorded against each query.

B. Dataset and Query Descriptions

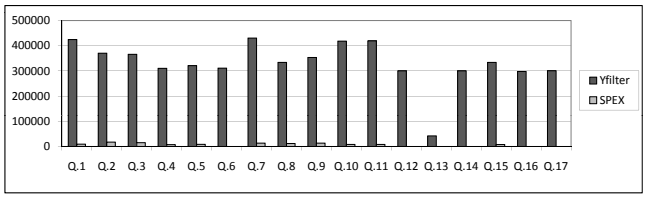
The dataset consists of the live streaming data recorded during a GAA Championship match. Our three datasets are of size 7MB, 14MB and 21MB respectively. The maximum depth of the XML data is eight levels. In order to evaluate the performance of the XML streaming engines SPEX and YFilter, the queries were designed to test each feature of the XPath language supported by the respective engines. The list of queries performed are displayed in Figure 6.



(a) Queries Performed on the 7MB Dataset



(b) Queries Performed on the 14MB Dataset



(c) Queries Performed on the 21MB dataset

Fig. 8. Query Performance over Three Datasets

1) Query Descriptions: Among the query set are queries containing nested paths, value predicates, attributes, the *text()* and *position()* functions and wildcards. The usage of the aforementioned features in these experiments was driven by user needs as determined by domain specialists. The axes examined were descendant-or-self and child. SPEX does not support attributes or the *position()* function, thus the queries numbered 6, 12-14 and 16-17 exploiting these features were run through the YFilter engine only. The grammar for each of the engines necessarily define the subset of the XPath language supported by that engine. We found the documented YFilter grammar to be accurate. The grammar defined for SPEX indicates support for the comparative operators when employing the *text()* function, however in our experiments we could not successfully replicate the $>$ or $<$ operators using *text()*, only the = and *!*= operators were functional.

2) Expanded Example:: SPEX accepts query input in the expanded XPath format only. Therefore we re-expressed the queries listed in Figure 6 in expanded form. Figure 7 displays queries numbered 5 and 10 expressed in expanded form.

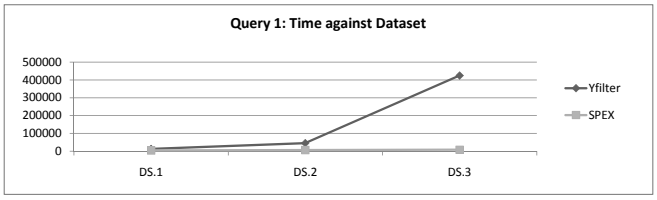


Fig. 9. Query 1 Performed over the Three Datasets

C. Performance Analysis

Figure 8 illustrates the performance of our query set over each of the three datasets. The X-axis indicates the query and the Y-axis identifies the time taken in milliseconds by the XML streaming engine to return the results of the query. It is evident from the graphs that SPEX outperforms YFilter in all cases. It is also evident that the factor by which SPEX is faster increases as the dataset gets larger. Figure 9 illustrates the increase in time taken to evaluate query number 1 over the three datasets. The time taken by SPEX to perform the query follows an approximately linear growth pattern, but the growth pattern for YFilter is approximately exponential. This correlation is typical for all queries over each of the datasets.

We observed that YFilter's support for the *position()* function is somewhat restricted. There may only be one instance of the *position()* function employed for any one element. For example, it is possible to return the first ten elements (*position()<=10*) but not the elements between position ten and twenty. Thus, range queries on a specific element with a user-defined lower and upper boundary are not possible. In addition the comparative operators in YFilter are restricted to the *position()* function, and cannot be used with *text()* values. Only equality can be performed on *text()* values. The boolean operators *and* and *or* are available in SPEX, but not in YFilter. The SPEX operator *and* when applied to predicates (illustrated as example 5B in Figure 7) has the same performance in terms of speed when rewriting the query with two predicates side by side (illustrated as example 5A in Figure 7).

D. Limitations

Although our query test suite listed in Figure 6 exploits a rich subset of the XPath query language, nevertheless they lack the full expressive syntax available in XPath that would permit *information-oriented* queries as opposed to *data-oriented* queries. The following query is a valid XPath expression for the data stream generated by our prototype wireless sensor device but is not supported by SPEX or YFilter.

> Display all 60-second periodic average heart rate readings above 120bpm that occur throughout the entire game where the actual heart rate reading is above 140bpm. This query will discard any sensor reading blips (or false readings) of heart values above 140bpm and only return real periods of strenuous activity.

```
//reading[key='Heart Rate'][value[.>140]]//average
[period[.=60000]]/value[.>120]
```

V. Conclusion

In this paper, motivated by the real-world deployment of a prototype wireless sensor vest in the sport science context, we addressed the requirements for a real-time streaming query engine capable of processing live data streams. We began by providing a thorough analysis and detailed evaluation of the existing state-of-the-art streaming query engines. We selected two of the most promising candidates with an implementation readily available and subjected them to a comprehensive suite of test queries designed to evaluate and benchmark both their capability as a streaming query engine and their suitability to process a given query according to its properties. Our results and subsequent analysis identified different scenarios and processing requirements whereby each of the engines provide a unique contribution. Thus, our conclusions motivate as future work the development of a filtering agent which, upon receiving a query and according to its properties, dynamically selects at runtime the appropriate streaming query engine to process the query. Indeed the filtering agent could occupy a key position as part of a more general query optimiser for XML streaming query engines over wireless sensor networks.

However, our experiments also highlighted the current immaturity of streaming query engines. In particular, the current implementations lack an expressive query syntax to fully exploit the sensors they query. The functionality and complexity available with the new and prototype wireless sensor devices and the data streams they generate are outpacing the streaming query engines' ability to exploit that functionality to the full. As such, there are key areas to be addressed in future work. These include the development of a richer query syntax and the faithful and more complete support for existing query languages such as XPath.

Acknowledgement

This work is partly supported by Science Foundation Ireland under grant 07/CE/I1147.

References

[1] M. Roantree, D. McCann, and N. Moyna, "Integrating Sensor Streams in pHealth Networks," in *ICPADS*, 2008, pp. 320–327.

[2] *XML Path Language (XPath) 2.0*, W3C Recommendation ed., World Wide Web Consortium, January 2007. [Online]. Available: http://www.w3.org/TR/xpath20/

[3] *XQuery 1.0: An XML Query Language*, W3C Recommendation ed., World Wide Web Consortium, January 2007. [Online]. Available: http://www.w3.org/TR/xquery/

[4] Simple API for XML (SAX 2.0). [Online]. Available: http://www.saxproject.org/

[5] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel, "The XML Stream Query Processor SPEX," in *ICDE*, 2005, pp. 1120–1121.

[6] D. Olteanu, T. Furche, and F. Bry, "Evaluating Complex Queries Against XML Streams with Polynomial Combined Complexity," in *BNCOD*, 2004, pp. 31–44.

[7] Y. Diao and M. J. Franklin, "High-Performance XML Filtering: An Overview of YFilter," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 41–48, 2003.

[8] YFilter User's Manual. [Online]. Available: http://yfilter.cs.umass.edu/html/manual/YFilter_User_Manual.html

[9] F. Peng and S. S. Chawathe, "XPath Queries on Streaming Data," in *SIGMOD Conference*, 2003, pp. 431–442.

[10] F. Peng and S. S. Chawathe, "XSQ: A Streaming XPath Engine," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 577–623, 2005.

[11] I. Avila-Campillo, T. J. Greeny, A. Gupta, M. Onizukaz, D. Raven, and D. Suciu, "XMLTK: An XML Toolkit for Scalable XML Stream Processing," *Proceedings of Programming Languages Technologies for XML*, 2002.

[12] S. Böttcher and R. Steinmetz, "Evaluating XPath Queries on XML Data Streams," in *BNCOD*, 2007, pp. 101–113.

[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer, "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 467–516, 2003.