Distributed Continuous Queries Over Web Service Event Streams

Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Austria
Email: {lastname}@infosys.tuwien.ac.at

***Abstract*—Complex Event Processing over Web service event streams poses huge challenges with regard to efficient, scalable execution as well as expressive models and languages that account for the dynamics in long-running queries. We present a distributed query platform that tackles these problems. Our novel query model permits to specify inputs that provide data for other inputs and need to be processed first. An XQuery language extension lets users easily express such dependencies, which are then continuously resolved with the required data at runtime. Query specifications are abstracted from physical deployment, allowing the platform to distribute the execution and to elastically scale up and down. We evaluate several aspects of our prototype in a Cloud computing environment.**

***Keywords*-complex event processing, continuous queries, scalability, elasticity, Cloud computing, Web services, WS-Eventing**

I. Introduction

Throughout the last years, the World Wide Web has moved from an Internet of documents to an Internet of services [1]. This goes hand-in-hand with the paradigm of Service-oriented Computing [2], which considers services as the building blocks for distributed applications. While the classical Web model is a client-server and request-response model, more and more emphasis is put on loosely coupled distributed systems, asynchronous processing and event-driven architectures [3]. WS-Eventing [4] has been proposed as a technology for building event-based services.

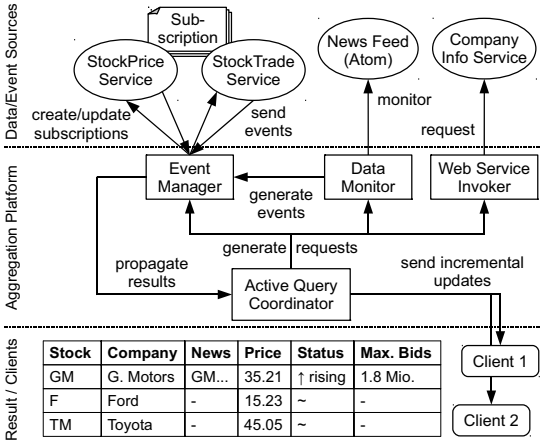
The primary format for data exchange on the Web is the Extensible Markup Language (XML). The XML query language XQuery [5] provides a powerful means to arbitrarily extract, transform and generate XML content. The current working draft of XQuery version 3.0 supports queries over sequences of XML nodes. This feature has been largely influenced by Botan et al. [6] who proposed to extend XQuery with *window functions*. These functions provide the basis for Complex Event Processing (CEP) [7] over XML event streams, i.e., sequences of temporally decoupled XML messages sent from a producer to subscribers. On top of XQuery, however, event subscription, collection, correlation and propagation still require tailor-made implementation on the application level. This inherent complexity is even harder to handle when the individual event streams have interdependencies, i.e., if a subscription needs to be updated when other event streams produce a certain result or pattern.

In earlier research, we have introduced the *WS-Aggregation* framework [8], a scalable platform which allows to combine and process data from heterogeneous Web resources using configurable distribution strategies. Whereas our previous work focused on synchronous, stateless aggregation of static Web data, we now extend the approach to support active (continuous) queries over dynamically changing data and event streams. This platform employs an novel active query model for event-based data aggregation, which not only processes events from a single source or subscription, but actively creates and renews event subscriptions based on user-defined data dependencies and generation templates. An aggregation query consists of an arbitrary number of inputs from Web services and allows to specify data dependencies between any two inputs i_1 and i_2. When i_1 yields a new result, the invocation or event subscription for i_2 is generated and renewed. The functional query specification is abstracted from its physical distribution, which allows to split the execution on multiple computing nodes for deployment in a scalable Cloud Computing environment.

In the remainder of this paper, we first introduce a scenario of continuous event querying in Section II, and provide some background on XQuery window clauses in Section III. Section IV presents the query model and discusses distributed query execution and runtime query updates. Section V details the platform implementation, and Section VI evaluates different aspects of the solution. In Section VII we discuss related work, and Section VIII concludes the paper.

II. Scenario

We consider a scenario from the financial computing domain, in which Web services provide live data about companies and stock prices. The aim is to combine the information in an XML document that is actively updated when the underlying data change. Figure 1 illustrates, on a high level, how data and events are received and processed.



Stock	Company	News	Price	Status	Max. Bids
GM	G. Motors	GM...	35.21	↑ rising	1.8 Mio.
F	Ford	-	15.23	~	-
TM	Toyota	-	45.05	~	-

Figure 1. Event-Based Continuous Data Aggregation Scenario

We distinguish 3 basic types of receiving data from the sources: (1) the *StockPrice* and *StockTrade* services allow to

subscribe for certain events transmitted using WS-Eventing, (2) the *News* feed is regularly monitored for changes, (3) the *Company Info* service contains rarely changing, static data. The clients specify a query to receive aggregated data that are incrementally updated as the query executes. The aggregation platform mediates between the data providers and consumers, and coordinates the query execution. The required core components from a high-level perspective are as follows. An Event Manager (EM) maintains subscriptions with the target services and receives events. The Web Service Invoker (SI) is responsible for performing synchronous service requests, and a Data Monitor (DM) repeatedly retrieves data from the monitored resource and generates an event to report any changes. The collected results from EM, DM and SI are fed into the Active Query Coordinator (AQC), which updates all dependencies and generates new requests as needed. To the clients the platform appears as a single entity, but in fact the system is distributed over several computing nodes, be it for performance reasons or due to higher-level constraints (e.g., the StockPrice and StockTrade services should report to physically separated endpoints).

The resulting document should contain a table with the current stock prices and general company information. Furthermore, the result indicates when a stock has three or more consecutive price rises, in which case the largest bid volume is displayed. If the platform detects that a stock has risen *and* traders are placing high volume bids (e.g., ≥ 1 million), the table should display live news about these companies.

A. Interdependent Event Streams

Figure 2 illustrates two sample event streams of the StockPrice and StockTrade services. Initially, a subscription for StockPrice exists, and the service continuously sends stock price events. When three consecutive rises are detected for *GM* (bold text), the Event Manager requests a new subscription with the StockTrade service to receive all *bids* and *asks* for *GM* placed on the market. Finally, this subscription is destroyed when five consecutive ticks (also in bold text) are below the last price of the rising sequence (35.7).

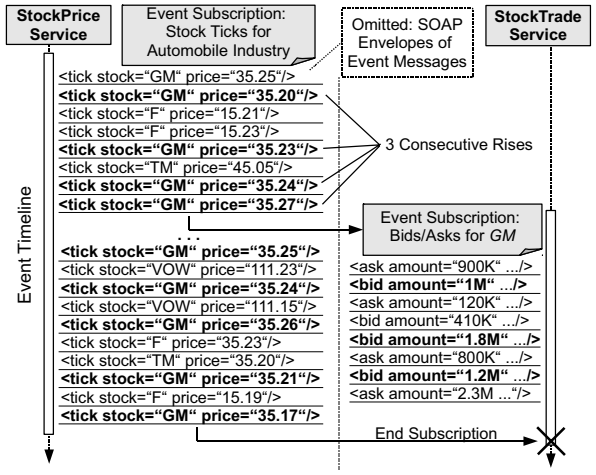


Figure 2. Sample Event Streams and Lifecycle of Event Subscriptions

The presented scenario poses several challenges to CEP. The basic requirement is the possibility to detect patterns in event streams in order to trigger new actions. This is a core research topic in CEP, and we can build on the existing work (e.g., [9], [10]). In this paper, we utilize XQuery to detect patterns in single streams, and perform distributed execution of continuous queries over multiple event sources. We thereby focus particularly on dependencies which determine the data flow and the lifecycle of event subscriptions.

III. Background

We briefly provide background information on XQuery window clauses, which is essential for understanding the remainder of the paper. The current draft of XQuery 3.0 introduces *tumbling* and *sliding window* clauses which we utilize as the basis for CEP. Listing 1 prints an exemplary window query operating on the StockTrade events. The variable `$input` points to the sequence of events that the platform received so far. The query creates a *window* `$w` (sequence of consecutive items drawn from the `$input` sequence) for each subsequence of `$input` for which the *start* and *end* conditions apply. A window starts when the bid amount is at least *1000000*, and the same window ends if a higher amount is found than in the previous window. In other words, the query splits the sequence `$input` into chunks of consecutive event sequences (stored in the loop variable `$w`), and returns the end item of each subsequence (which is greater than the end item of the preceding subsequence).

```
1  for tumbling window $w in $input/bid
2  start $s at $spos previous $sprev when
3     number($s/@amount) ge 1000000
4  end $e next $enext when $spos le 1 or
5     number($sprev/@amount) lt number($e/@amount)
6  return
7  <maxbid stock="{$e/@stock}">{$e/@amount}</maxbid>
```

Listing 1. XQuery Window Clause for Events from StockTrade Service

The defining characteristic of tumbling windows is that new windows are only created when the previous window has been closed, whereas sliding windows may overlap [6]. The two window types provide orthogonal functionality and it depends on the application which of the two is used. Note that XQuery provides these features for queries over event streams, but has no explicit means for active continuous queries that consider data dependencies between streams, automatically request and retrieve data from services, and manage the lifecycle of event subscriptions. Section IV discusses how this is achieved by WS-Aggregation.

IV. Event-Based Web Data Aggregation

In this section we detail our approach for event-based querying and aggregation of Web services and data, discussing both the functional aspects (i.e., how clients express aggregation queries), and the non-functional aspects which affect the platform's internal structure and mode of operation. We present the model that is used to construct aggregation queries in Section IV-A. The actual query language, presented in Section IV-B, is based on XQuery and introduces

additional language constructs that are tailored to the used query model. The WS-Aggregation platform is particularly designed for scalability and Cloud-based deployment, which will be discussed in more detail in Section IV-C.

A. Query Model

A simplified version of the query model is illustrated as a UML class diagram in Figure 3. The model builds on our previous work [8] and extends it with eventing-specific aspects. The central entity *AggregationQuery* specifies the Endpoint Reference (EPR) used to receive result updates (*notifyTo*). An aggregation query contains multiple *Input*s (identified by *ID*) that determine how data from external sources are retrieved and inserted into the active query.

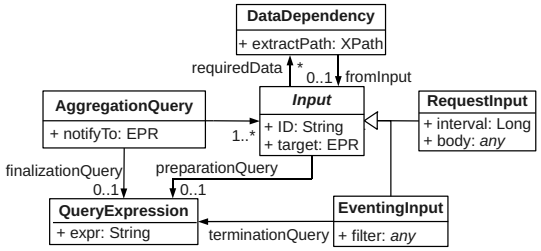


Figure 3. Query Model for Continuous Event-Based Data Aggregation

The *EventingInput* entity creates event subscriptions with an optional *filter* that is evaluated by the target Web service as defined in WS-Addressing. On the other hand, *RequestInput* is used for documents retrieved in a request-response manner. In both cases, the *target* EPR specifies the location of the service. The *interval* attribute allows to continuously monitor a Web service or document for changes.

Besides input entities, an aggregation query contains XQuery-based *QueryExpression*s. A *preparationQuery* expression may be used to prepare and transform the result of an Input immediately as it arrives at the platform. In the case of a RequestInput, the preparation query performs a one-time transformation (e.g., extracting the news for a certain company from the News Feed), whereas to "prepare" an EventingInput a window query is continuously executed on the event stream to yield new results. To specify the condition for ending an event subscription, an EventingInput is associated with a *terminationQuery*. When this query yields a *true* result, the target service is automatically invoked with a WS-Eventing *Unsubscribe* message to destroy the subscription. Finally, the *finalizationQuery* combines all the prepared results and constructs the final output document.

B. Input Data Dependencies

A core feature in the query model is the concept of data dependencies between two inputs i_1 and i_2, which signifies that i_2 can only be "activated" if certain data from i_1 are available to be inserted into i_2. Activation in this context means that the input becomes usable only when all data dependencies are resolved. The query model in Figure 3 associates an Input (*receiving* input), via the association class *DataDependency*, with an arbitrary number of required data from other Inputs (*providing* inputs) of the same query. The attribute *extractPath* is an XPath which points to the data in the providing input. If the optional association *fromInput* is set, the data will be extracted from a specific Input; otherwise, if *fromInput* is unknown, the platform continuously matches *extractPath* against the available inputs and extracts data when this XPath evaluates to *true*.

```
[new] DataDependency ::= "$" Name? "{" PathExpr "}"
[new] EscapeDollar ::= "$$"
[125] PrimaryExpr ::= DataDependency | ...
[145] CommonContent ::= DataDependency | EscapeDollar | ...
[204] ElementContentChar ::= Char - [{}<&$]
[205] QuotAttrContentChar ::= Char - ["{}<&$]
[206] AposAttrContentChar ::= Char - ['{}<&$]
```

Listing 2. XQuery Language Extension for Data Dependencies

We propose an XQuery language extension to account for simple modeling of data dependencies as studied in this paper. The modifications are printed in EBNF (Extended Backus-Naur Form) syntax in Listing 2. The new construct is named *DataDependency* and consists of a dollar sign ("$"), an optional *Name* token referencing the *ID* of the providing input, and an XPath expression (*PathExpr*) specifying the *extractPath* in curly brackets ("{", "}"). To express that a string "`${foo}`" should be interpreted as a verbatim string and not as a data dependency, a double dollar sign (*EscapeDollar*) is used for escaping ("`$${foo}`"). The *DataDependency* token is added to the definition of *PrimaryExpr* (rule 125 in the current version of XQuery 3.0) and *CommonContent* (rule 145). Furthermore, to satisfy parser consistency of the syntax rules, the single dollar sign needs to be appended to the list of "exceptional" (non-content) characters (rules 204 to 206). Section V provides further implementation details.

1) Scenario Query Model: Based on the four inputs for the scenario services (SP = StockPrice, ST = StockTrade, CN = Company News, CI = Company Information), the expressions in Figure 4 illustrate how data dependencies (highlighted in bold font) can be utilized in different parts of an aggregation query, including the WS-Eventing *filter* for the ST subscriptions (input 2), the *preparationQuery* for the monitored CN Feed (input 3), and the SOAP body of the CI request (input 4). The *terminationQuery* of ST is similar to its *preparationQuery* (see Listing 1) and not printed in the figure. The input results are directly accessible in the *finalizationQuery*, e.g., `//news` inserts the results of input 3 into the final result. The figure depicts the *Dependency Graph* indicating which input expects data from which other inputs. This graph is automatically constructed from the aggregation query and checked for circular dependencies.

It is important to notice that the XQuery expressions are evaluated in a preprocessing step which generates the actual elements to be used as, e.g., *filter* for ST and *body* for CI. For instance, if during evaluation of `$1{//rising/@stock}` the platform extracts three stock symbols ('GM','F','TM') from the prepared result of input 1, three instances of input 2 are generated for each of the corresponding filter

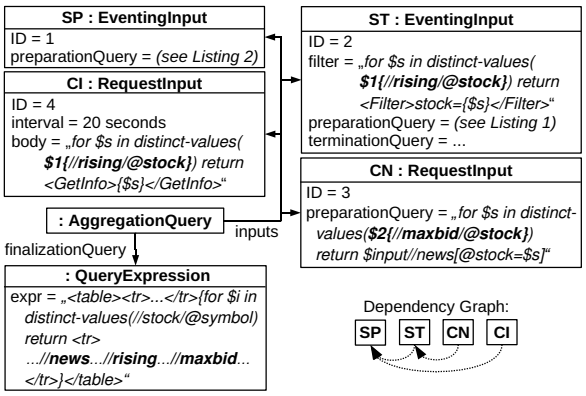


Figure 4. Aggregation Query for Data Aggregation Scenario

expressions. Analogously, three instances of input 4 are generated with corresponding `GetInfo` service requests. Similarly, the *preparationQuery* of CN loops over all `maxbid` stock symbols from input 2 (cf. preparation query in Listing 1) and outputs all news lines related to these symbols.

C. Distributed Query Execution

To serve a large number of simultaneous active queries, the platform employs a scalable distributed processing model with several loosely coupled nodes working collaboratively. In addition to the obvious performance reasons, query distribution may also be required or desired from a higher-level (business) perspective. For instance, the data may have to be physically separated according to business policies. Moreover, if multiple data sources are spread over a large geographical distance, the aggregation can be organized in a location-based hierarchical structure, e.g., with regional and national nodes (for more details see [8]).

The fundamental assumption of WS-Aggregation is therefore that multiple aggregator machines collaboratively process the queries and events requested by the clients. The set of available aggregators is stored in a central service registry, which allows to dynamically select a subset of aggregators responsible for executing each individual query. The overall request is then split up into smaller (“atomic”) parts that can be processed by a single node (*generateRequests* function in Algorithm 1). It should be stated that the single parts cannot be regarded as completely isolated units, because, according to the query model, there often exist data dependencies between them. Each time a new event is received and added to the result store (*onEvent* function), the dependencies are updated and possibly new request inputs are generated. Note that several inputs, possibly from different aggregation queries, can be affected by an event in the *onEvent* function.

Line 6 in function *generateRequests* indicates that a responsible aggregator is determined for each input. WS-Aggregation supports different configurable distribution strategies, and allows to either specify fixed input-to-aggregator mappings or to assign inputs automatically. In the latter case, the platform performs load balancing. In general,

Algorithm 1 Processing of Active Query with Dependencies

$results \leftarrow$ new result store // variable for aggregation results
function generateRequests(AggregationQuery r)
1: **while** r contains independent inputs **do**
2: $I \leftarrow$ determine independent inputs in r
3: **for all** $i \in I$ **do**
4: $G \leftarrow$ generate actual inputs from i using XQuery engine
5: **for all** $input \in G$ **do**
6: $aggr \leftarrow$ determine aggregator to handle $input$
7: **if** $aggr$ is self **then**
8: $result \leftarrow$ invoke $input$ on $input.target$
9: $result \leftarrow$ apply preparation query to $result$
10: add $result$ to $results$, update dependencies
11: **else**
12: delegate request with $input$ to $aggr$
13: **end if**
14: **end for**
15: **end for**
16: **end while**
function onEvent(Event e) // called by WS-Eventing service
1: add e to event buffer of e
2: **for all** EventingInput i affected by e **do**
3: $result \leftarrow$ apply preparation query of i to event buffer of e
4: add $result$ to $results$, notify clients, update dependencies
5: generateRequests($i.aggregationQuery$) // issue new requests
6: **end for**

new inputs are assigned to aggregators with the lowest load (CPU, memory, number of active queries). A second important distribution goal for event-based processing is to bundle inputs with the same underlying event stream. Consider two inputs i_1 and i_2 which receive the same ticks from StockPrice, but use a different preparation query to filter certain information. If these inputs are handled by some aggregator a, a shared event buffer can be used and redundancies are avoided to save memory. Of course, this approach does not scale infinitely, and inputs are assigned to new aggregators if the load of a reaches a certain threshold. The evaluation in Section VI further discusses this aspect.

V. Implementation

This section discusses our implementation of the presented approach. WS-Aggregation employs multiple aggregator nodes which collaboratively implement the functionality of the aggregation platform as sketched earlier in Figure 1. From an external viewpoint, an aggregator is solely defined by its Web service interfaces. Specialized implementations can be plugged into the platform, as long as the aggregator registers itself in the *Service Registry*. The WS-Eventing compliant *Eventing Interface* is used to receive events from data services. the *Event Store* buffers and forwards the events to the *Query Engine* which consists of the *Preprocessor* (responsible for processing the XQuery data dependency extensions) and a third-party (hence depicted in gray) *XQuery Engine* from *http://mxquery.org*. The *Active Query Coordinator* (AQC), accessible from the *Aggregation Interface*, maintains aggregation queries, determines which data dependencies are fulfilled and activates new inputs.

The AQC forwards activated inputs to the *Request Distributor*, which implements configurable query distribution

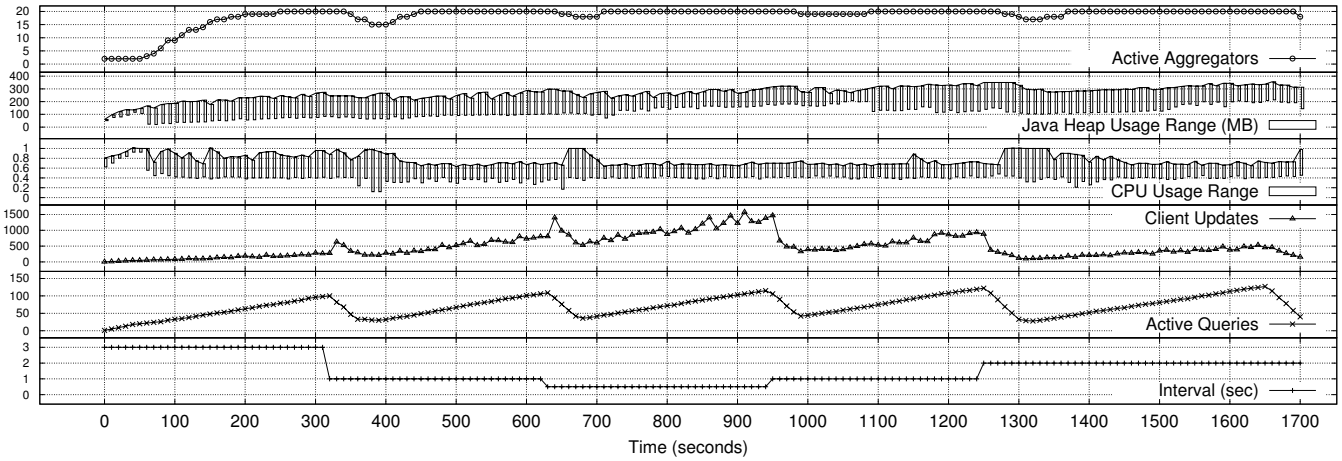


Figure 5. Performance Results of Executing Multiple Simultaneous Scenario Aggregation Queries with Different Event Frequencies

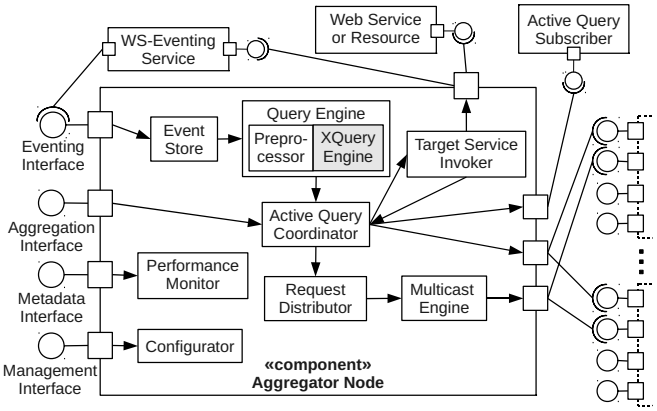


Figure 6. Core Components and Connectors of Aggregator Nodes

strategies. Moreover, the AQC uses the Eventing Interface of partner aggregators to propagate *WindowQueryEvent*s. To communicate with other nodes, the Request Distributor makes use of the *Multicast Engine*, which contacts the Aggregation Interface (to delegate the execution of inputs) or *Metadata Interface* of the partners. Results are pushed to the clients (Active Query Subscribers) by the AQC; alternatively, clients can poll for results (e.g., useful for Web browsers).

The implementation of the Preprocessor and XQuery language extension for data dependencies is based on JavaCC, a parser generator for Java. The EBNF syntax rules of XQuery were extended with the modifications in Listing 2 and transformed into the format of JavaCC. The parser generated by JavaCC reads in the extended XQuery expressions and creates an in-memory representation (abstract syntax tree), which is used to extract the data dependencies.

VI. Evaluation

To evaluate of the framework performance, we have set up a comprehensive test environment in the Amazon Elastic Compute Cloud (EC2). We launched an initial number of 15 aggregator nodes. During execution, the framework was configured to deploy up to 5 additional instances, which is achieved using the Web services based API of EC2. Furthermore, we deployed the four scenario Web services which provide randomized test data, and a *gateway* service which acts as a central entry point for clients and selects master aggregators for coordinating the query execution.

Figure 5 illustrates the results of the scenario execution. The x-axis shows the time in seconds. The lowermost part of the figure plots the interval in which the test StockPrice and StockTrade services publish events to the platform. Over time, various test clients deployed in a LAN outside of EC2 (average latency of 60ms) have requested and terminated multiple (up to 125 simultaneous) executions of the scenario aggregation in different variants (sub-plot *Active Queries*). The number of *Client Updates* per ten seconds (up to 1500 around time point 900) is largely influenced by a combination of event *Interval* (between 0.5 and 3 sec.) and Active Queries, and also depends on the random test data and the state of each active query.

Heap memory and CPU are shown in Figure 5 with the range (min. to max.) and the trendline of the maximum over all active aggregators. The platform heuristically distributes the total load, based on CPU/memory usage, active aggregators and queries. Up to second 50, the queries are handled by only two aggregators, because the aim is to bundle queries for one event stream on the same aggregator. However, after 60 seconds, additional aggregators are involved to avoid performance deterioration due to the increasing load. When ten aggregators are active, the platform requests new machines in addition to the 15 initial instances. The startup (roughly 40 sec.) includes EC2 overhead and the time to initialize and add the aggregator to the registry. As the active queries decrease, some aggregators become idle (e.g., time 400); the configurable timeout for releasing unused resources should be at least several minutes because aggregators may become used again (e.g., time 400) and instances are billed per hour.

We observe that the load is stable and equally distributed (small CPU and memory ranges) when the number of active queries only slightly changes (e.g., between time 400-600

or 700-900); however, rapid changes in the active queries cause load peaks, as event stores are initialized/terminated and many objects need to be allocated/freed. Note that the memory consumption grows particularly at the beginning, because the nodes perform internal caching. A factor that evidently raises memory management issues is the need to store past events for evaluation of query windows. Fortunately, MXQuery employs sophisticated algorithms to free unused input buffer items, and we have run several complex queries with up to 1 million events without memory leaks.

VII. Related Work

The broad range of applications for event processing has spurred the interest of both industry and research for this topic [11]. In the context of service-oriented computing, some work on bringing the power of CEP into service environments has been carried out within the VRESCo project [12]. However, so far eventing is mostly used for monitoring (e.g., [13]), while WS-Aggregation uses the CEP notion for service-based data aggregation. Related to the data aggregation aspect is the *Qizx* XML database engine [14], which provides an XQuery implementation supporting most 3.0 features. As WS-Aggregation, Qizx Server performs data management tasks on demand, however, it lacks support for complex event processing and WS-Eventing. Similarly, the *Active XML* project [15] provides distributed data management and different styles of data integration. However, ActiveXML follows a top-down approach, whereas WS-Aggregation employs a bottom-up query model, in which atomic inputs of an aggregation query are executed and correlated to dynamically compose the final result. Another related problem area is distributed filtering of XML documents, e.g., XFilter [16]. XFilter is an XPath based approach to structural document filtering. A similar contribution has been presented in [17], where nondeterministic finite automata have been distributed over Pastry distributed hashtables in order to filter XML. Both approaches have a strong focus on high-performance XML filtering, while WS-Aggregation provides a much larger feature set. Other approaches to distributed event processing generally sacrifice query expressiveness and Web standards for performance. For instance, S4 [18] is a stream computing platform that focuses on scalable and fault tolerant processing of massive numbers of events. Our distributed query processing approach has also been influenced by the method in [19] which proposes stratums as a way to achieve modularization and distribution. Finally, our work in WS-Aggregation is related to the field of Enterprise Information Integration (EII) [20].

VIII. Conclusion

We presented a platform for active event-based aggregation of Web data. The active query model utilizes XQuery window clauses, and provides a language extension to model data dependencies between event streams or other query inputs. The system is designed for scalability and distributed query execution, and allows easy deployment in the Cloud. As part of our ongoing work, we are investigating advanced techniques for optimized load distribution, bundling multiple queries on shared event buffers, and further adoption of Autonomic Computing [21] concepts to the platform's control loops. We also envision a light-weight abstraction of XQuery clauses to account for often recurring CEP patterns.

Acknowledgments

This research was funded by the European Community's Seventh Framework Programme under grant agreement 257483 (Indenica).

References

[1] C. Schroth and T. Janner, "Web 2.0 and SOA: Converging Concepts Enabling the Internet of Services," *IT Professional*, vol. 9, no. 3, 2007.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *IEEE Computer*, vol. 40, 2007.

[3] H. Taylor, A. Yochem, L. Phillips, and F. Martinez, *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*, 1st ed. Addison-Wesley Professional, 2009.

[4] W3C, "Web Services Eventing (WS-Eventing)," http://www.w3.org/Submission/WS-Eventing/, 2006.

[5] ——, "XQuery 1.0: An XML Query Language," http://www.w3.org/TR/xquery/, 2007.

[6] I. Botan, D. Kossmann, P. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius, "Extending XQuery with window functions," in *VLDB*, 2007.

[7] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications Co., 2010.

[8] W. Hummer, P. Leitner, and S. Dustdar, "WS-Aggregation: Distributed Aggregation of Web Services Data," in *SAC*, 2011.

[9] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient Pattern Matching Over Event Streams," in *ACM SIGMOD*, 2008, pp. 147–160.

[10] P. Fischer, A. Garg, and K. S. Esmaili, "Extending XQuery with a pattern matching facility," in *Int. XML Database Symposium*, 2010, pp. 48–57.

[11] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[12] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Advanced Event Processing and Notifications in Service Runtime Environments," in *DEBS*, 2008, pp. 115–125.

[13] E. Mulo, U. Zdun, and S. Dustdar, "Monitoring Web Service Event Trails for Business Compliance," in *SOCA*, 2009.

[14] Pixware, "Qizx, a fast XML database engine fully supporting XQuery," http://www.xmlmind.com/qizx/.

[15] S. Abiteboul, O. Benjelloun, and T. Milo, "The Active XML project: an overview," *VLDB Journal*, vol. 17, 2008.

[16] M. Altinel and M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," in *VLDB*, 2000, pp. 53–64.

[17] I. Miliaraki and M. Koubarakis, "Distributed Structural and Value XML Filtering," in *DEBS*, 2010, pp. 2–13.

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *ICDM Workshops*, 2010, pp. 170–177.

[19] G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion, "A stratified approach for supporting high throughput event processing applications," in *DEBS*, 2009, pp. 1–12.

[20] P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Communications of the ACM*, vol. 51, 2008.

[21] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41 – 50, 2003.