# Continuously Updating Query Results over Real-Time Linked Data

Ruben Taelman, Ruben Verborgh, Pieter Colpaert,
Erik Mannens, and Rik Van de Walle

Data Science Lab (Ghent University - iMinds)
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{firstname.lastname}@ugent.be

**Abstract.** Existing solutions to query dynamic Linked Data sources extend the SPARQL language, and require continuous server processing for each query. Traditional SPARQL endpoints accept highly expressive queries, contributing to high server cost. Extending these endpoints for time-sensitive queries increases the server cost even further. To make continuous querying over real-time Linked Data more affordable, we extend the low-cost Triple Pattern Fragments (TPF) interface with support for time-sensitive queries. In this paper, we discuss a framework on top of TPF that allows clients to execute SPARQL queries with continuously updating results. Our experiments indicate that this extension significantly lowers the server complexity. The trade-off is an increase in the execution time per query. We prove that by moving the complexity of continuously evaluating real-time queries over Linked Data to the clients and thus increasing the bandwidth usage, the cost of server-side interfaces is significantly reduced. Our results show that this solution makes real-time querying more scalable in terms of CPU usage for a large amount of concurrent clients when compared to the alternatives.

**Keywords:** Linked Data, Linked Data Fragments, SPARQL, continuous querying, real-time querying

## 1 Introduction

As the Web of Data is a *dynamic* dataspace, different results may be returned depending on when a question was asked. The end-user might be interested in seeing the query results update over time, and thus may have to re-execute the entire query over and over again (i.e., polling). This is, however, not very practical, especially if it is unknown beforehand when data will change. An additional problem is that many public (even static) SPARQL query endpoints suffer from a low availability [5]. The unrestricted complexity of SPARQL queries [15] combined with the public character of SPARQL endpoints entails a high server cost, which makes it expensive to host such an interface with high availability. Dynamic SPARQL streaming solutions offer combined access to dynamic data streams and static background data through continuously executing queries. Because of this continuous querying, the cost for these servers is *even higher* than with static querying.

In this work, we therefore devise a solution that enables clients to continuously evaluate non-high frequency queries by polling specific fragments of the data. The

resulting framework performs this without the server needing to remember any client state. Its mechanism requires the server to *annotate* its data so that the client can efficiently determine when to retrieve fresh data. The generic approach in this paper is applied to the use case of public transit route planning. It can be used in various other domains with continuously updating data, such as smart city dashboards, business intelligence, or sensor networks.

In the next section, we discuss related research on which our solution will be based. In Section 3, we present a motivating use case. Section 4 discusses different techniques to represent dynamic data, after which Section 5 gives an explanation of our proposed query solution. Next, Section 6 shows an overview of our experimental setup and its results. Finally, Section 7 discusses the conclusions of this work with further research opportunities.

## 2 Related Work

In this section, we first explain techniques to perform RDF annotation, which will be used to determine freshness. Then, we zoom in on possible representations of temporal data in RDF. We finish by discussing existing SPARQL streaming extensions and a low-cost (static) Linked Data publication technique.

### 2.1 RDF Annotations

Annotations allow us to attach metadata to triples. We might for example want to say that a triple is only valid between a certain time interval, or that a triple is only valid in a certain geographical area.

RDF 1.0 [11] allows triple annotation through *reification*. This mechanism uses *subject*, *predicate* and *object* as predicates, which allow the addition of annotations to such reified RDF triples. The downside of this approach is that one triple is now transformed to three triples, which significantly increases the total amount of triples.

*Singleton Properties* [14] create unique instances (singletons) of predicates, which then can be used for further specifying that relationship by for example adding annotations. New instances of predicates are created by relating them to the old predicate through the `sp:singletonPropertyOf` predicate. While this approach requires fewer triples than reification to represent the same information, it still has the issue of the original triple being lost, because the predicate is changed in this approach.

With RDF 1.1 [6] came *graph* support, which allows triples to be encapsulated into named graphs, which can also be annotated. Graph-based annotation requires fewer triples than both reification and singleton properties when representing the same information. It only requires the addition of a fourth element to the triple which transforms it to a quad. This fourth element, the *graph*, can be used to add the annotations to.

### 2.2 Temporal data in the RDF model

Regular RDF triples cannot express the time and space in which the fact they describe is true. In domains where data needs to be represented for certain times or time ranges,

these traditional representations should thus be extended. *Time labeling* [9] is the process of annotating triples with their change time, as opposed to making completely new snapshots of the graph every time a change occurs. Gutierrez et al. made a distinction between *point-based* and *interval-based* labeling, which are interchangeable [8]. The former states information about an element at a certain time instant, while the latter states information at all possible times between two time instants.

The authors introduced a *temporal vocabulary* [8] for the discussed mechanisms, which will be referred to as `tmp` in the remainder of this document. Its core predicates are:

**`tmp:interval`** This predicate can be used on a subject to make it valid in a certain time interval. The range of this property is a time interval, which is represented by the two mandatory properties `tmp:initial` and `tmp:final`.

**`tmp:instant`** Used on subjects to make it valid on a certain time instant as a point-based time representation. The range of this property is `xsd:dateTime`.

**`tmp:initial` and `tmp:final`** The domain of these predicates is a time interval. Their range is a `xsd:dateTime`, and they respectively indicate the start and the end of the interval-based time representation.

Next to these properties, we will also introduce our own predicate `tmp:expiration` with range `xsd:dateTime` which indicates that the subject is only valid up until the given time.

### 2.3 SPARQL Streaming Extensions

Several SPARQL extensions exist that enable querying over data streams. These data streams are traditionaly represented as a monotonically non-decreasing stream of triples that are annotated with their timestamp. These require *continuous processing* [7] of queries because of the constantly changing data.

*C-SPARQL* [4] is an approach to querying over static and dynamic data. This system requires the client to *register* a query to the server in an extended SPARQL syntax that allows the use of *windows* over dynamic data. This *query registration* [3, 7] must occur by clients to make sure that the streaming-enabled SPARQL endpoint can continuously re-evaluate this query, as opposed to traditional endpoints where the query is evaluated only once. C-SPARQL's execution of queries is based on the combination of a regular SPARQL engine with a *Data Stream Management System* (DSMS) [2]. The internal model of C-SPARQL creates queries that distribute work between the DSMS and the SPARQL engine to respectively process the dynamic and static data.

*CQELS* [12] is a "white box" approach, as opposed to "black box" approaches like C-SPARQL. This means that CQELS natively implements all query operators without transforming it to another language, removing the overhead of delegating it to another system. According to previous research [12], CQELS performs much better than C-SPARQL for large datasets; for simple queries and small datasets the opposite is true.

### 2.4 Triple Pattern Fragments

Experiments have shown that more than half of public SPARQL endpoints have an availability of less than 95% [5]. A possible contributor to this low availability is the

unrestricted complexity of SPARQL queries [15] combined with the unpredictable user and query load of SPARQL endpoints. Clients can send arbitrarily complex SPARQL queries, which could form a bottleneck in endpoints. *Triple Pattern Fragments* (TPF) [17] aim to solve this issue of high interface cost by moving part of the query processing to the client, which reduces the server load at the cost of increased query times and bandwidth. The endpoints are limited to an interface through which only separate triple patterns can be queried instead of full SPARQL queries. The client is then responsible for carrying out the remaining evaluation of complex SPARQL queries asked by its users.

## 3 Use Case

A guiding use case, based on public transport, will be referred to in the remainder of this paper. When public transport route planning applications return dynamic data, they can account for factors such as train delays as part of a continuously updating route plan. In this use case, different clients need to obtain all train departure information for a certain station. A static SPARQL query can be used to retrieve all information using this basic data model, which can be found in Listing 1.1, the first two triple patterns refer to triples that contain dynamic data.

```
SELECT ?delay ?platform ?headSign ?routeLabel ?departureTime
WHERE {
    _:id t:delay         ?delay.
    _:id t:platform      ?platform.
    _:id t:departureTime ?departureTime.
    _:id t:headSign      ?headSign.
    _:id t:routeLabel    ?routeLabel.
    FILTER (?departureTime > "2015-12-08T10:20:00"^^xsd:dateTime).
    FILTER (?departureTime < "2015-12-08T11:20:00"^^xsd:dateTime).
}
```

**Listing 1.1:** The basic SPARQL query for retrieving all upcoming train departure information in a certain station. The two first triple patterns are dynamic, the last three are static.

## 4 Dynamic Data Representation

Our solution consists of a partial redistribution of query evaluation workload from the server to the client, which requires the client to be able to access the server data. There needs to be a distinction between regular static data and continuously updating dynamic data in the server's dataset. For this, we chose to define a certain temporal range in which these dynamic facts are valid, as a consequence the client will know when the data becomes invalid and has to fetch new data to remain up-to-date. To capture the temporal scope of data triples, we annotate this data with time. In this section, we discuss two different types of time labeling, and different methods to annotate this data.

### 4.1 Time Labeling Types

We use interval-based labeling to indicate the *start and endpoint* of the period during which triples are valid. Point-based labeling is used to indicate the *expiration time*.

With expiration times, we only save the latest version of a given fact in a dataset, assuming that the old version can be removed when a newer one arrives. These expiration times provide enough information to determine when a certain fact becomes invalid in time. We use time intervals for storing multiple versions of the same fact, i.e., for maintaining a history of facts. These time intervals must indicate a start- and endtime for making it possible to distinguish between different versions of a certain fact. These intervals cannot overlap in time for the same facts.

### 4.2 Methods for Time Annotation

The two time labeling types introduced in the last section can be annotated on triples in different ways. In Section 2.1 we discussed several methods for RDF annotation. We will apply time labels to triples using the singleton properties, graphs and implicit graphs annotation techniques.

**Singleton Properties** *Singleton properties* annotation is done by creating a singleton property for the predicate of each dynamic triple. Each of these singleton properties can then be annotated with its time annotation, being either a time interval or expiration times.

**Graphs** To time-annotate triples using *graphs*, we can encapsulate triples inside contexts, and annotate each context graph with a time annotation.

**Implicit Graphs** A TPF interface gives a unique URI to each fragment corresponding to a triple pattern, including patterns without variables, i.e., actual triples. Since Triple Pattern Fragments [17] are the basis of our solution, we can interpret each fragment as a graph. We will refer to these as *implicit graphs*. This URI can then be used as graph identifier for this triple for adding time information. For example, the URI for the triple <s> <p> <o> on the TPF interface located at `http://example.org/dataset/` is `http://example.org/dataset?subject=s&predicate=p&object=o`.

We will execute our use case for each of these annotation methods. In practice, an annotation method must be chosen depending on the requirements and available technologies. Each has their own set of advantages and disadvantages, like for example required triple count.

## 5 Query Engine

TPF query evaluation involves server and client software, because the client actively takes part in the query evaluation, as opposed to traditional SPARQL endpoints where the server does all of the work. Our solution allows users to send a normal SPARQL query to the local query engine which autonomously detects the dynamic parts of the query and continuously sends back results from that query to the user.

Our solution must be able to handle regular SPARQL 1.1 queries, detect the dynamic parts, and produce continuously updating results for non-high frequency queries. To achieve this, we built an extra software layer on top of the existing TPF client that supports each discussed labeling type and annotation method and is capable of doing dynamic query transformation and result streaming. We will refer to this layer as the *query streamer*. A simplified overview of this architecture can be seen in Figure 1. The *rewriter* module internally splits the query up in a static and dynamic query. This splitting is done by exchanging metadata with the endpoint using small queries. The dynamic query is formed in such a way that time annotations can be retrieved from its dynamic triples based on the selected annotation mechanism. The *streamer* module does the result streaming by determining the earliest expiration time of the dynamic triples for each resultset and re-executing the dynamic query based on that time. Once the dynamic query results are collected, these results are filled into the static query after which it can be executed. These dynamic and static results are then combined and sent back to the user. A client-side caching mechanism is applied in the streamer module to cache the static query results where possible. At the TPF server, dynamic data must be annotated with time depending on the used combination of labeling type and method. The server expects dynamic data to be pushed to the platform by an external process with varying data. For the case of graph-based annotation, we extended the TPF server implementation with quad support.

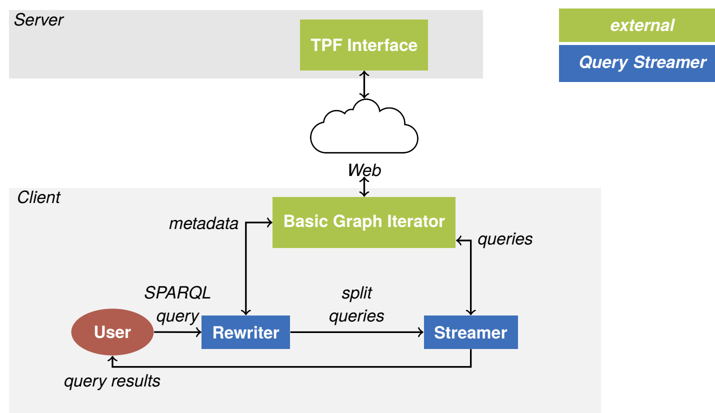


**Fig. 1:** Overview of the proposed client-server architecture.

## 6 Evaluation

We set up an experiment to measure the impact of our proposed redistribution of workload between the client and server by simultaneously executing a set of queries against a server using our proposed solution. We repeat this experiment for two state-of-the-art solutions: C-SPARQL and CQELS.

To test the client and server performance, our experiment consisted of one server and ten physical clients. Each of these clients can execute from one to twenty unique

concurrent queries based on the use case from Section 3. The data for this experiment was derived from real-world Belgian railway data using the iRail API[1]. This results in a series of 10 to 200 concurrent query executions. This setup was used to test the client and server performance of different SPARQL streaming approaches.

For comparing the efficiency of different time annotation methods and for measuring the effectiveness of our client-side cache, we measured the execution times of the query for our use case from Section 3. This measurement was done for different annotation methods, once with the cache and once without the cache. For discovering the evolution of the query evaluation efficiency through time, the measurements were done over each query stream iteration of the query.

The discussed architecture was implemented[2] in JavaScript using Node.js to allow for easy communication with the existing TPF client.

The tests[3] were executed on the Virtual Wall (generation 2) environment from iMinds [10]. Each machine had two Hexacore Intel E5645 (2.4GHz) CPUs with 24 GB RAM and was running Ubuntu 12.04 LTS. For CQELS, we used version 1.0.1 of the engine [13]. For C-SPARQL, this was version 0.9 [16]. The dataset for this use case consisted of about 300 static triples, and around 200 dynamic triples that were created and removed each ten seconds. Even this relatively small dataset size already reveals important differences in server and client cost, as we will discuss in the paragraphs below.

**Server Cost** The server performance results from our main experiment can be seen in Figure 2a. This plot shows a quickly increasing CPU usage for C-SPARQL and CQELS for higher numbers of concurrent query executions. On the other hand, our solution never reaches more than one percent of server CPU usage.

**Client Cost** The results for the average CPU usage across the duration of the query evaluation of all clients that sent queries to the server in our main experiment can be seen in Figure 2b. The clients that were sending C-SPARQL and CQELS queries to the server had a client CPU usage of nearly zero percent for the whole duration of the query evaluation. The clients using the client-side query streamer solution that was presented in this work had an initial CPU peak reaching about 80%, which dropped to about 5% after 4 seconds.

**Annotation Methods** The execution times for the different annotation methods can be seen in Figure 3. The execution times for expiration time annotation used are approximately constant, while the execution times for time intervals significantly increase over time, for graph-based annotation this increase is not as fast as for the other two approaches.

[1] https://hello.irail.be/api/1-0/

[2] The source code for this implementation is available at https://github.com/rubensworks/TPFStreamingQueryExecutor

[3] The code used to run these experiments with the relevant queries can be found at https://github.com/rubensworks/TPFStreamingQueryExecutor-experiments/

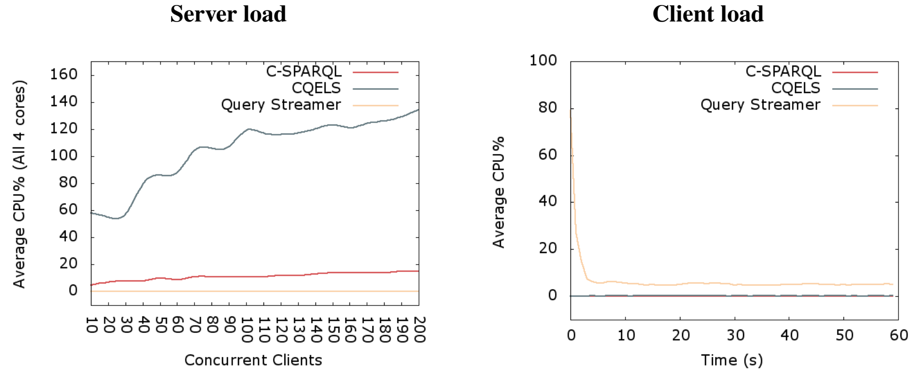


**Fig. a:** The server CPU usage of our solution proves to be influenced less by the number of clients.

**Fig. b:** In the case of 200 concurrent clients, client CPU usage initially is high after which it converges to about 5%. The usage for C-SPARQL and CQELS is almost non-existing.

**Fig. 2:** Average server and client CPU usage for one query stream for C-SPARQL, CQELS and the proposed solution. Our solution effectively moves complexity from the server to the client.

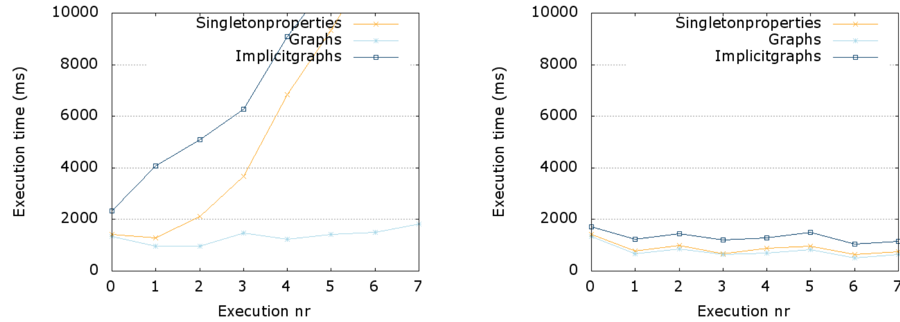


**Fig. a:** Time intervals.

**Fig. b:** Expiration times.

**Fig. 3:** Executions times for the three different types of dynamic data representation for several subsequent streaming requests. The figures show a mostly linear increase when using time intervals and constant execution times for annotation using expiration times. They also reveal that the graph approach has the lowest execution times.

## 7 Conclusions

In this paper, we researched a solution for querying over dynamic data with a low server cost, by continuously polling the data based on volatility information. In this section, we draw conclusions from our evaluation results with regard to server cost, client cost, performance, and annotation methods.

**Server cost** Not only is the server cost for our solution more than ten times lower on average when compared to the alternatives, this cost also increases much slower for a growing number of simultaneous clients. This makes our proposed solution more scalable for the server for our tested use case.

**Client cost** The results for the client load measurements from Section 6 show that our solution requires more client processing power than C-SPARQL and CQELS The required client processing power using our solution is clearly much higher than for C-SPARQL and CQELS. This is because we redistributed the required processing power from the server to the client. In our solution, it is the client that has to do most of the work for evaluating queries, which puts less load on the server. The load on the client still remains around 5% for the largest part of the query evaluation as shown in Figure 2b. Only during the first few seconds, the query engines CPU usage peaks, which is because of the processor-intensive rewriting step that needs to be done once at the start of each dynamic query evaluation.

**Performance** Our solution significantly reduces the required server processing per client, this complexity is mostly moved to the client. This comparison shows that our technique allows data providers to offer dynamic data which can be used to continuously evaluate dynamic queries with a low server cost. Our low-cost publication technique for dynamic data is useful when the number of potential simultaneous clients is large. When this data is only required for a small number of clients in a closed off environment and query evaluation must happen fast, traditional approaches like CQELS or C-SPARQL are advised. These are only two possible points on the *Linked Data Fragments* axis [17], depending on the publication requirements, combinations of these approaches can be used.

**Annotation methods** From the results in Section 6, we can conclude that graph-based annotation results in the lowest execution times. It can also be seen that annotation with time intervals has the problem of continuously increasing execution times, because of the continuously growing dataset. Time interval annotation can be desired if we for example want to maintain the history of certain facts, as opposed to just having the last version of facts using expiration times. In future work, we will investigate alternative techniques to support time interval annotation without the continuously increasing execution times.

In this work, the frequency at which our queries are updated is purely data-driven using time intervals or expiration times. In the future it might be interesting, to provide a control to the user to change this frequency, if for example this user only desires query updates at a much lower frequency than the data actually changes.

In future work, it is important to test this approach with a larger variety of use cases. The time annotation mechanisms we use are generic enough to transform all static facts to dynamic data for any number of triples. The CityBench [1] RSP engine benchmark can for example be used to evaluate these different cases based on city sensor data. These tests must be scaled (both in terms of clients as in terms of dataset size), so that the maximum number of concurrent requests can be determined, with respect to the dataset size.

## Acknowledgments

The described research activities were funded by iMinds and Ghent University, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union. Ruben Verborgh is a Postdoctoral Fellow of the Research Foundation Flanders.

## References

1. Ali, M.I., Gao, F., Mileo, A.: CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets. In: The Semantic Web - ISWC 2015, Lecture Notes in Computer Science, vol. 9367, pp. 374–389 (2015)
2. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford data stream management system. Book chapter (2004)
3. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Stream Reasoning: Where We Got So Far. In: Proceedings of the NeFoRS2010 Workshop (2010)
4. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Rec. 39(1), 20–26 (Sep 2010)
5. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL web-querying infrastructure: Ready for action? In: The Semantic Web–ISWC 2013, pp. 277–293 (2013)
6. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and abstract syntax. Recommendation, W3C (Feb 2014), http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
7. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! Reasoning upon rapidly changing information. Intelligent Systems, IEEE 24(6), 83–89 (Nov 2009)
8. Gutierrez, C., Hurtado, C., Vaisman, A.: Introducing time into RDF. Knowledge and Data Engineering, IEEE Transactions on 19(2), 207–218 (Feb 2007)
9. Gutierrez, C., Hurtado, C., Vaisman, A.: Temporal RDF. In: The Semantic Web: Research and Applications, pp. 93–107 (2005)
10. iLab.t, iMinds: Virtual Wall: wired networks and applications, http://ilabt.iminds.be/virtualwall
11. Klyne, G., J. Carroll, J.: Resource Description Framework (RDF): Concepts and abstract syntax. Rec., W3C (Feb 2004), http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
12. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and Linked Data. In: The Semantic Web–ISWC 2011, pp. 370–388 (2011)
13. Levan, C.: CQELS engine: Instructions on experimenting CQELS, https://code.google.com/p/cqels/wiki/CQELS_engine
14. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like RDF reification? Making statements about statements using singleton property. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 759–770. WWW '14, New York, NY, USA (2014)
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: International semantic web conference. vol. 4273, pp. 30–43 (2006)
16. StreamReasoning: Continuous SPARQL (C-SPARQL) ready to go pack, http://streamreasoning.org/download
17. Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., Van de Walle, R.: Querying datasets on the Web with high availability. In: Proceedings of the 13th International Semantic Web Conference (2014)