Scheduling for SPARQL Endpoints

Fadi Maali, Islam A. Hassan, and Stefan Decker

Insight Centre for Data Analytics, National University of Ireland Galway
{firstname.lastname}@insight-centre.org

Abstract. When providing public access to data on the Semantic Web, publishers have various options that include downloadable dumps, Web APIs, and SPARQL endpoints. Each of these methods is most suitable for particular scenarios. SPARQL provides the richest access capabilities and is the most suitable option when granular access to the data is needed. However, SPARQL expressivity comes at the expense of high evaluation cost. The potentially large variance in the cost of different SPARQL queries makes guaranteeing consistently good quality of service a very difficult task. Current practices to enhance the reliability of SPARQL endpoints, such as query timeouts and limiting the number of results returned, are far from ideal. They can result in under utilisation of resources by rejecting some queries even when the available resources are sitting idle and they do not isolate "well-behaved" users from "ill-behaved" ones and do not ensure fair sharing among different users. In similar scenarios, where unpredictable contention for resources exists, scheduling algorithms have proven to be effective and to significantly enhance the allocation of resources. To the best of our knowledge, using scheduling algorithms to organise query execution at SPARQL endpoints has not been studied. In this paper, we study, and evaluate through simulation, the applicability of a few algorithms to scheduling queries received at a SPARQL endpoint.

1 Introduction

When providing public access to data on the Semantic Web, publishers have various options that include downloadable dumps, Web APIs, and SPARQL endpoints. Each of these methods is most suitable for particular scenarios, however none of them provides an ideal global solution [7, 13]. SPARQL, the recommended W3C query language[1], is an attractive option to provide expressive access to RDF data. SPARQL is basically a graph pattern matching language that provides rich capabilities for slicing and dicing RDF data. The latest version, SPARQL 1.1, added support for aggregation, nested and distributed queries, and other features.

However, supporting public SPARQL access to data is expensive. It has been shown that evaluating SPARQL is PSPACE-complete in general and coNP-complete for well-defined queries [10]. Therefore, the cost of different SPARQL

[1] http://www.w3.org/TR/sparql11-query/

queries can vary a lot; making guaranteeing consistently good quality of service a very difficult task. Evidence of this can be seen on the SPARQL Endpoint Status web page[2], in literature [2] and across the Web[3] and the blogosphere[4].

Existing SPARQL endpoints employ different measures to enhance their reliability and to ensure consistent quality of service. Such measures include query timeouts, refusing expensive SPARQL queries, limiting the number of triples returned or returning partial results. For example, 4Store supports a soft limit for execution time[5] and Virtuoso allows setting a maximum threshhold on the expected query cost[6]. There are still a number of problems with these approaches: (i) they provide an inconsistent user experience and limit the expressiveness of allowed queries (ii) there is no clear way to communicate these non-standard shortcomings to the user (iii) they can result in under utilisation of resources by rejecting some queries even when the available resources are sitting idle (iv) they do not isolate "well-behaved" users from "ill-behaved" ones and do not ensure fair sharing among different users.

In similar scenarios, where unpredictable contention for resources exists, scheduling algorithms have proven to be effective and to significantly enhance the allocation of resources. Scheduling has been utilised for data networks [3, 12], for processes assignment in operating systems [8], for cloud and grid computing [9, 4], and recently to schedule jobs sent to a Hadoop cluster [15, 14]. Nevertheless, to the best of our knowledge, it has not been studied in the context of SPARQL endpoints.

In this paper we argue for employing scheduling algorithms to organise query execution at a, possibly public, SPARQL endpoint. Giving the wide applicability of scheduling, there exists a large number of scheduling algorithms. In this paper, we study, and evaluate through a simulation, the applicability of a few algorithms to schedule queries received at some SPARQL endpoint. A number of scheduling algorithms, mainly from the data networks domain, are reviewd (Section 3.1). We then describe their applicability to scheduling SPARQL queries (Section 3.2) and study through a simulation their effect on two popular SPARQL engines (Section 4).

We do not claim that scheduling solves the problem of providing a reliable publicly-accessible SPARQL endpoint. Nevertheless, our results show that scheduling enhances throughput and reduces the effect of complex queries on simpler ones. We also note that scheduling has the extra advantages of rewarding socially-aware behaviour and achieving better utilisation and fairer allocation of the available resources.

---

[2] `http://sparqles.okfn.org/`

[3] See for example `http://answers.semanticweb.com/questions/14440/can-the-economic-problem-of-shared-sparql-endpoints-be-solved`

[4] E.g. `http://daverog.wordpress.com/2013/06/04/the-enduring-myth-of-the-sparql-endpoint/` and `http://ruben.verborgh.org/blog/2013/09/30/can-i-sparql-your-endpoint/`

[5] `http://4store.org/trac/wiki/SparqlServer`

[6] `http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtSPARQLEndpointProtection`

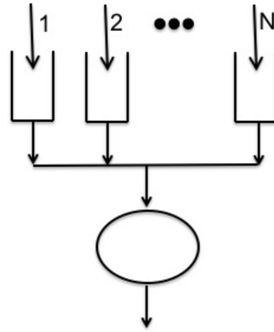


Fig. 1: General model of scheduling

2 Related Work

Most current practices to enhance the reliability of SPARQL endpoints are based on introducing ad-hoc measures and limits such as query timeouts, refusing expensive SPARQL queries, limiting the number of triples returned or returning partial results. These measures have a number of problems as discussed in the introduction. Linked Data Fragments [13] is a recent proposal to enhance the reliability of SPARQL endpoints via shifting part of the computation needed to answer SPARQL queries towards the client side. Furthermore, [1] proposed a vision for a workload-aware and adaptive system to deal with the diversity and dynamism that are inherent in SPARQL workloads. To the best of our knowledge, scheduling has not been studied in the context of SPARQL endpoints. Nevertheless, scheduling has been used and studied in many domains.

Scheduling has been extensively studied in data networks where the capacity of a switch is shared by multiple senders [3, 6, 12]. We describe the main algorithms used in data networks in the next section.

Scheduling has also been used to organise jobs of shared Hadoop clusters. First In First Out (FIFO) Scheduler, Fair Scheduler[7] and Capacity Scheduler[8] are the most widely used schedulers in practice. Similar to our work, these schedulers re-use algorithms defined for the data networks. Notice that, in contrast to SPARQL queries, Hadoop jobs are expected to take a long time and their execution can be pre-empted.

Moreover, scheduling has also been applied in wireless networks [5], grid computing [4] and distributed hash tables [11].

3 Scheduling

Figure 1 depicts a generic model for scheduling. There is a single server, and N job arrival streams, each feeding a different first in, first out (FIFO) queue.

[7] http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html
[8] http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html

The scheduler decides which stream gets served at each moment and for how long. The goal is usually to maximise throughput while ensuring fair sharing of resources and avoiding long waiting times. Keeping a separate queue for each stream puts up "firewalls" protecting well-behaved streams against streams that might otherwise saturate the server capacity and drive delays to unacceptable levels.

3.1 Scheduling Algorithms

We next describe a number of scheduling algorithms used mainly for data networks and then discuss their applicability to SPARQL endpoints.

Ideal Scheduler A mathematical idealization of queuing, which was originally proposed as an idealization of time-slicing in computer systems (Kleinrock 1976 as cited in [6]), known as Processor Sharing (PS). In PS, the server cycles through the active jobs, giving each a small time quantum of service, and then preempting the job to work on the next. The PS mechanism allots resources fairly and provides full utilisation of the resources. However, in settings where pre-empting jobs is not feasible, such as data networks, PS cannot be applied. A number of "emulations" of it exist nevertheless. We next discuss two of them.

Fair Scheduler Described in [3] and [6]. Fair scheduling emulates the PS fair scheduling by maintaining a notion of virtual time (what time an input stream would have been served had PS been applied). Streams are then served in increasing order of their virtual finish time. It has been shown that this algorithm emulates the fair PS algorithm well [6].

Deficit Round-Robin Scheduler Described in [12]. Each input stream holds a deficit counter (a credit balance) and only gets served if the query cost is less than its balance. After execution, the cost is subtracted from the balance. If the queue is not served due to insufficient balance, it gets a quantum charge that can be used in the next round, i.e. a queue is compensated when its service is delayed.

3.2 Scheduling for SPARQL

We consider scheduling as a service running on top of SPARQL endpoints. The scheduler receives the queries and then decides in what order to send them to the endpoint. The goal is to: (i) minimize the effect that expensive queries can have on other queries (ii) maximize the utilization of the available resources (iii) reward socially-aware behaviour (e.g., simpler queries that set a limit on the number of required results).

However, contrary to the typical scheduling scenario depicted in Figure 1, a SPARQL endpoint can handle multiple queries at a time. In fact, as triple

stores, Web servers and the underlying operating systems each have their own resource utilisation and sharing facilities, it will be very inefficient to send only one query at a time to the endpoint. Therefore, SPARQL scheduler sends multiple queries to the endpoint as long as their total cost is under a configured threshold. Upon the completion of processing some query, its cost is substracted from the tracked total cost. The threshold is necessary to avoid overwhelming the endpoint as sending many queries simultaneously to an endpoint results in rejecting to process many queries by the endpoint.

In practice, three further challenges need to be addressed:

- Efficiently estimating the cost of an incoming SPARQL query. The cost involves multiple paramters such as CPU, memory, network and I/O cost. The cost also depends on the query, the data and the triple store. However, this problem is beyond the scope of this paper.
- Identifying the source of each query in order to define streams of inputs. In the absence of user authentication, IP addresses and session detection techniques can be used.
- Setting the threshold of total computing capacity. This can be set via experimenting.

4 Simulation Experiment

4.1 Implementation

We implemented three scheduling algorithms in Java [9]:

- **FIFO**: A First-In-First-Out queue. This scheduler serves input streams in order (one query at a time) while keeping track of the total cost of queries being processed at every point and ensuring that this cost is always kept lower than the computing capacity threshold.
- **Deficit**: Implements a deficit round-robin algorithm as described in Section 3.1.
- **Fair**: Implements a fair scheduling algorithm as described in Section 3.1.

4.2 Experiment Setup

We experimented with two triple stores, Virtuoso Open-Source Edition 7.0.0 Release[10] and Jena Fuseki 1.0.1[11]. Both triple stores were run on a Mac with 8GB memory and a 2.9GHz Intel Core i7 CPU. Jena Fuseki was run in memory with a maximum heap size of 3GB.

We used the Semantic Web Dog Food[12] data that contains information about papers published in the main conferences and workshops in the area of Semantic

[9] The code is avaialble at https://gitlab.insight-centre.org/Maali/sparql-endpoints-scheduler
[10] http://virtuoso.openlinksw.com/
[11] http://jena.apache.org/documentation/serving_data/
[12] http://data.semanticweb.org/

Web research. The data was downloaded[13] from the Semantic Web Dog Food website in Februaury 2014.

The Web logs of the Semantic Web Dog Food were made available as part of the USEWOD 2013 Data Challenge[14]. To get real-world SPARQL queries for our experiment, we extracted SPARQL queries from these Web logs. We ran each query three times on Fuseki (without any added scheduling) and classified it as simple or complex based on the time it took. All simple queries took an average of less than 90ms, while complex queries took more than 450ms. We use these numbers (90 and 450) as a proxy for the query costs.

We simulated two concurrent flows of queries, one with simple queries and the other with complex queries. This simulates two applications with different needs. Queries were selected randomly from the set of simple and complex queries respectively. Both flows follow a Poisson distribution. We experimented with differnet means of the Poisson distribution (a.k.a. interval) and with different thesholds for the computing capacity (i.e., total cost of queries being processed at a time). Each run was stopped after 5 minutes and logs were collected then.

We consider a query to be fully processed if it is sent to the endpoint and all results sent back from the endpoint are received (within the 5 minute cut-off). For each query we measure two durations:

Waiting Time: the time taken from the moment the query is received at the scheduler until it is sent to the endpoint.

Processing Time: the time taken from the moment the query is received at the scheduler until the moment at which the query is fully processed.

We wanted to include a no-scheduling setting as a baseline, however running these flows without any scheduling resulted in overwhelming the endpoint and therefore most of the queries were dropped without getting answered.

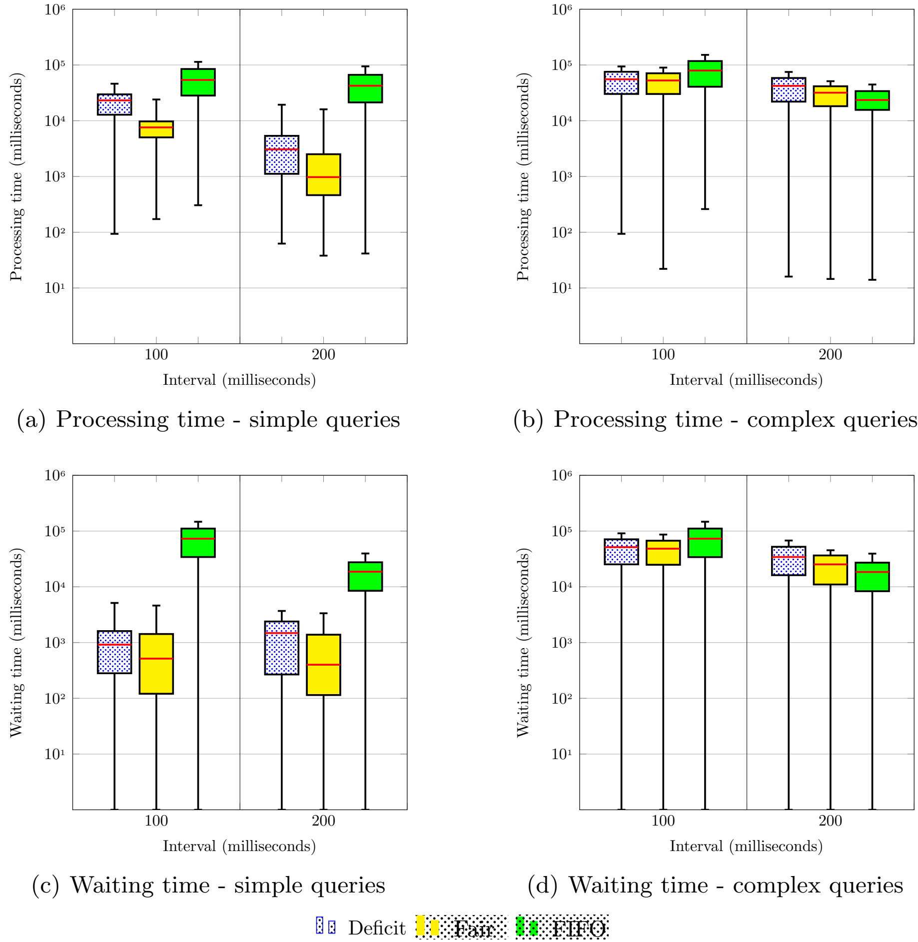
4.3 Results & Discussion

We experimented with intervals of 100, 200, 500 and 1000 milliseconds (ms). Queries sent with intervals 500 and 1000 ms were not frequent enough to require any waiting and therefore resulted in no scheduling. We report only on queries with intervals 100 and 200 ms. We experimented with different thresholds for the computing capacity. We report here only on the results when the threshold is set to 8000 because it showed the most effective results for the three scheduling algorithms tested; all larger values overwhelmed the endpoint. Notice that 8000 is just a proxy for the computation capacity available and it needs to be interpreted together with the cost of the queries (we used 90 and 450 for simple and complex queries respectively).

Table 1 shows the percentage of queries completed when sent at a 100 ms interval on Fuseki. On Virtuoso all queries were fully processed, while on Fuseki it can be noticed that the FIFO algorithm processed equivalent percentages of

[13] `http://data.semanticweb.org/dumps/`

[14] `http://data.semanticweb.org/usewod/2013/challenge.html`



(a) Processing time - simple queries

(b) Processing time - complex queries

(c) Waiting time - simple queries

(d) Waiting time - complex queries

Fig. 2: Virtuoso - Processing and waiting times

both simple and complex queries while the other two algorithms “favoured” simple queries and “penalised” complex ones. Deficit scheduling showed better throughput.

Figures 2 and Figure 3 show the processing and waiting times for simple and complex queries when running against Virtuoso and Fuseki. Similar to the throughput results, it can be noticed how penalising complex queries results in smaller waiting and processing times for simple queries when using deficit or fair scheduling. On the other hand, complex queries have smaller waiting and processing times using FIFO scheduling with 200 milliseconds interval. The higher processing time of complex queries using FIFO scheduling was surprising. Our interpretaion of this is that prioritising simple queries allowed better utilisa-

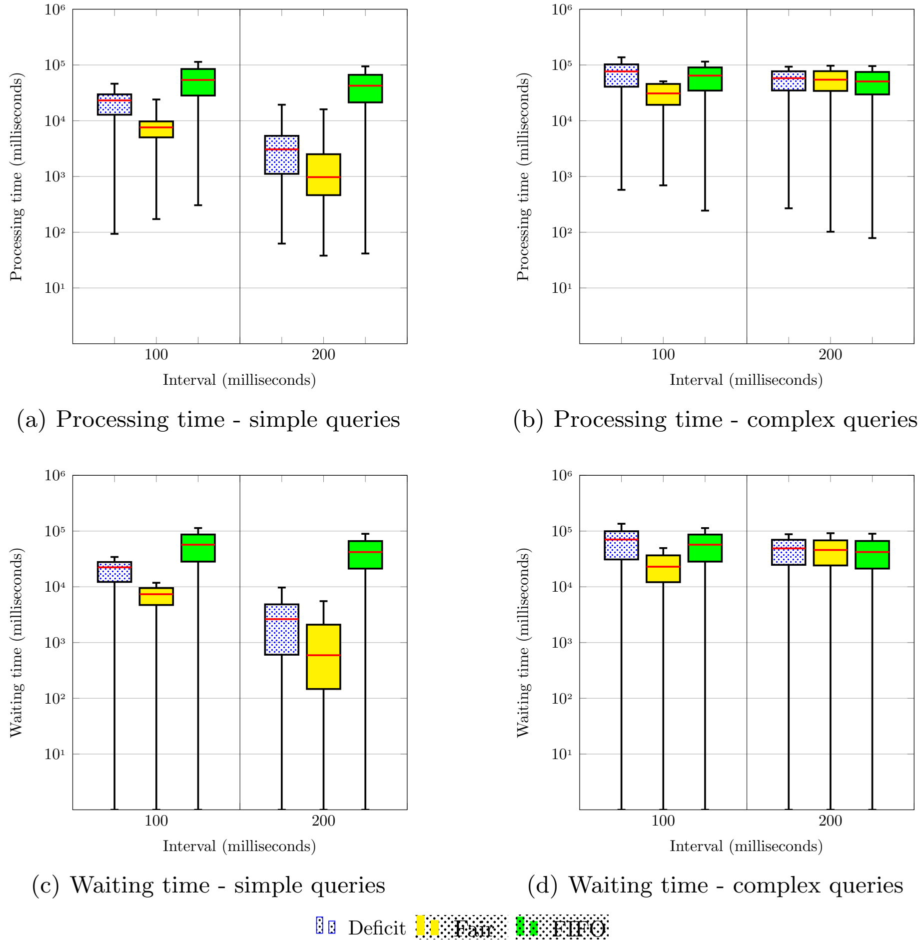


Fig. 3: Fuseki - Processing and waiting times

tion of the available resources when queries arrive at a high frequency that can overwhelm the endpoint.

In summary, scheduling allowed getting better throughput by delaying queries instead of rejecting them (recall that running without scheduling resulted in rejecting most of the queries). Deficit and Fair scheduling algorithms favoured simpler queries. In general, deficit scheduling, the simpler algorithm, showed better results in our settings than fair scheduling.

	Simple Queries	Complex Queries
FIFO	47.3%	47.4%
Deficit	100%	42%
Fair	77%	40%

Table 1: Percentage of fully processed queries on Fuseki (throughput)

5 Conclusions & Future Work

We reported a simulation experiment to study the effects different scheduling algorithms can have when used to organise execution of SPARQL queries received at some endpoint. We note that simple scheduling can be implemented with minimal overhead and has effect only when queries are received at high frequency.

We consider this work as an initial step and hope to extend the experiment and deploy it in some real-world use case.

Acknowledgements. Fadi Maali is funded by the Irish Research Council, Embark Postgraduate Scholarship Scheme. This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

References

1. G. Aluc, M. T. Özsu, and K. Daudjee. Workload matters: Why rdf databases need a new design. *PVLDB*, 7(10):837–840, 2014.
2. C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *ISWC 2013*, pages 277–293. Springer, 2013.
3. A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.
4. F. Dong and S. G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. *School of Computing, Queen's University, Kingston, Ontario*, 2006.
5. H. Fattah and C. Leung. An Overview of Scheduling Algorithms in Wireless Multimedia Networks. *Wireless Communications, IEEE*, 9(5):76–83, 2002.
6. A. G. Greenberg and N. Madras. How fair is fair queuing. *Journal of the ACM (JACM)*, 39(3):568–598, 1992.
7. A. Hogan and C. Gutierrez. Paths towards the Sustainable Consumption of Semantic Data on the Web. In *AMW*, 2014.
8. L. Kleinrock. Queueing systems, volume II: Computer applications. 1976.
9. H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis. Schedule Optimization for Data Processing Flows on the Cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 289–300. ACM, 2011.
10. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

11. S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and its Uses. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 73–84. ACM, 2005.
12. M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Networking, IEEE/ACM Transactions on*, 4(3):375–385, 1996.
13. R. Verborgh, M. Vander Sande, P. Colpaert, S. Coppens, E. Mannens, and R. Van de Walle. Web-scale Querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*, 2014.
14. M. Yong, N. Garegrat, and S. Mohan. Towards a Resource Aware Scheduler in Hadoop. In *Proc. ICWS*, pages 102–109, 2009.
15. M. Zaharia. Job scheduling with the fair and capacity schedulers. *Hadoop Summit*, 9, 2009.