

Scalable Linked Data Stream Processing via Network-Aware Workload Scheduling

Lorenz Fischer, Thomas Scharrenbach, Abraham Bernstein

University of Zurich, Switzerland*
{lfischer,scharrenbach,bernstein}@ifi.uzh.ch

Abstract. In order to cope with the ever-increasing data volume, distributed stream processing systems have been proposed. To ensure scalability most distributed systems partition the data and distribute the workload among multiple machines. This approach does, however, raise the question how the data and the workload should be partitioned and distributed. A uniform scheduling strategy—a uniform distribution of computation load among available machines—typically used by stream processing systems, disregards network-load as one of the major bottlenecks for throughput resulting in an immense load in terms of inter-machine communication.

In this paper we propose a graph-partitioning based approach for workload scheduling within stream processing systems. We implemented a distributed triple-stream processing engine on top of the Storm realtime computation framework and evaluate its communication behavior using two real-world datasets. We show that the application of graph partitioning algorithms can decrease inter-machine communication substantially (by 40% to 99%) whilst maintaining an even workload distribution, even using very limited data statistics. We also find that processing RDF data as single triples at a time rather than graph fragments (containing multiple triples), may decrease throughput indicating the usefulness of semantics.

Keywords: semantic flow processing, stream processing, linked data, complex event processing, graph partitioning, workload scheduling

1 Introduction

In today’s connected world, data is produced in ever-increasing volume, velocity, variety, and veracity [20]: sensor data is gathered, transactions are made in the financial domain, people post/tweet messages, humans and machine infer new information, etc. This phenomenon can also be found on the Web of Data (WoD), where new sources are made available as linked data. In order to process these

* The research leading to these results has received funding from the Europ. Union 7th Framework Programme FP7/2007-2011 under grant agreement no 296126 and from the Dept. of the Navy under Grant NICOP N62909-11-1-7065 issued by Office of Naval Research Global.

growing data sources many have proposed the use of distributed infrastructures such as Hadoop [20]. The batch-oriented synchronous nature of these solutions, however, may not be suited to ensure the timeliness of data processing. To address this shortcoming stream processing approaches based on information-flow processing have been proposed [8]. These systems continuously ingest new data as it arrives and process it online rather than storing it for batch-like processing. This continuous processing puts a significant load on employed systems and is, obviously, limited by the capacity of the employed hardware infrastructure.

To cope with increasing demands distributed stream processing systems have been proposed, which usually ensure scalability by partitioning the data and distributing the processing thereof to multiple machines. Note that deciding how to partition the data and distribute the associated processing is a non-trivial task and can have dire consequences on performance.

Distributed processes communicate with each other by sending messages containing partial results of the overall processing task. Processes on different machines communicate over the network and the resulting network load limits scalability in two ways: First, network traffic is several orders of magnitude slower than in-machine communication.¹ Second, the bandwidth of each machine limits the amount it can possibly communicate to processes residing on other machines. *As a consequence, finding a good distribution strategy for distributed stream processing is crucial to ensure scalability.* Note that the variety and potential schemalessness of linked data further aggravates the problem as a Semantic Flow Processing (SFP) systems (1) cannot rely on the schema for data partitioning and distribution and (2) the triple-nature (rather than the reliance on n-tuples or relations) of the underlying data model potentially further subdivides the smallest unit of data increasing the number of possible partitioning (and hence, distributions).

As distributed stream processing becomes more important in many areas of business and science, researchers have proposed various ways to schedule workload in such systems. Interestingly, we found no previous work that employs existing graph partitioning algorithms to the problem of workload scheduling.

In this paper, *we propose the use of graph partitioning algorithms to optimize the assignment of tasks to machines*: we regard the data-flow within an SFP system as a graph, where the edges' weight represents the required bandwidth for information passing and the vertices' weight represents the computational load required to process incoming messages. Specifically, we operationalize the edge weights as the number of messages sent from one process to another, based on the two assumptions: first, that messages are approximately the same size, and second the computational load to be proportional to the number of messages received by a processing vertex, assuming further that all messages need the same time to be processed on all tasks. We then use a graph partitioning algorithm to optimize the distribution of processes to machines whilst addressing two possibly

¹ Numbers from 2009: 500 times for latency, 40 times for bandwidth. See <http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf> for more details.

opposing goals: we try to minimize costly network messages whilst distributing computation load as evenly as possible.

To evaluate our distribution approach we implemented an SFP system on top of the Storm realtime computation framework.² Our implementation allows compiling certain SPARQL queries, modified for stream processing, to a Storm topology. To enable parallelization of the processing functions the data flows in the topology are partitioned using appropriate hash functions. Next we empirically determine the weight of the nodes and edges, partition the graph, and “schedule” the processes accordingly on machines. In our evaluation using two real-world datasets we show that our graph-partitioning strategy can decrease inter-machine communication substantially (by 40% to 99%) whilst maintaining an even workload distribution, even using very limited data statistics.

The remainder of this paper is structured as follows: next we succinctly summarize the most relevant related literature (Section 2) before introducing the details of our approach (Section 3). This is followed by a description of our experiments (Section 4), a discussion of the results in the light of its limitations (Section 5), and an exploration of the implications and future work (Section 6).

2 Related Work

This study relates to (distributed) Information Flow Processing (IFP)³, Semantic Flow Processing (SFP) [18], and workload scheduling. We provide an overview of the most relevant work in these fields.

Information Flow Processing: The field of IFP is vast and a survey is beyond the scope of this paper [15, 12, 8]. Here we only discuss the Aurora/Borealis [1, 2] systems, as they are most closely related to our research.

Aurora [1] lets the user specify a query using a set of operators (boxes), which are connected by links (arrows). The *Borealis* system [2] extends Aurora to include—among other things—the distribution of the workload across multiple machines. Load distribution is achieved by *query partitioning* – the assignment of operators (boxes) or a collection thereof (superboxes) to worker machines. This approach has two drawbacks: First, it limits the degree of parallelism to the number of boxes. Second, in its naïve setup, all information exchange between operators goes over the network consuming enormous amounts of network bandwidth. The only improvement to this strategy is to group operators onto the same machine. Our approach, in contrast, proposes to improve load distribution through *data partitioning*, where operators themselves are replicated across many machines.

Semantic Flow Processing: The *C-SPARQL* system [6] performs query matching on subsets of the information flow defined by windows. For query matching on the subsets it uses a regular SPARQL engine that is extended by some stream related

² <http://storm-project.net>

³ The term Information Flow Processing has been suggested by [8] as the term *Stream Processing* is ambiguous due to its usage by both the Complex Event Processing (CEP) and the Data Stream Management Systems (DSMS) community.

features. A distributed version was implemented using Apache S4 platform;⁴ yet with no particular scheduling strategy [9].

EP-SPARQL [4] is a complex event processing system, which extends the ETALIS system with a flow-extension of SPARQL for query processing [4]. ETALIS is a Prolog engine for which no distributed version exists yet.

CQELS [13] “implements the required query operators natively to avoid the overhead and limitations of closed system regimes”. It optimizes the execution by dynamically re-ordering operators to “prune the triples that will not make it to the final output” thus limiting processing. As the implementation makes no assumptions about scheduling with regards to messages sent between algebra components CQELS could benefit from a scheduler based on graph partitioning.

SPARQL_{Stream}[7] is a streaming extension to SPARQL that allow users to query relational data streams over a set of stream-to-ontology mappings. The language supports powerful windowing constructs and *SRBench* [23] uses it as the default engine for evaluation.

INSTANS [17] and *Sparkwave*[11] are based on a RETE network. Both their implementations are non-distributed implementation, yet very efficient. Both stream querying systems support the RDF, and RDFS, and—in the case of Sparkwave—OWL entailment. It would be interesting to investigate, to what extent our approach could be built on top of a RETE-network.

Workload Scheduling: Earlier work on scheduling in stream processing concentrated on operator scheduling in wide area networks [16] and admission control [21, 22], recent work also targets the usecase in which workload of a stream processor in a compute cluster has to be scheduled [5].

SODA [21] is an admission control system and task scheduler for System S [3]. The task scheduler within SODA is based on a mixed-integer optimization program and also uses techniques from the network flow literature.

Pietzbuch et al. [16] present an decentralized algorithm that is geared towards minimizing the overall latency of a stream processor whose operators are spread out across a wide area network, while taking CPU load into account.

Xia et al. [22] map the problem of task scheduling to a multicommodity flow network and present a distributed scheduling algorithm in which the amount of communication between nodes is incrementally analyzed and reduced.

Aniello et al. [5] present two algorithms which are both geared towards reducing the number tuples transferred over the network of a storm cluster. Their static “offline” scheduler takes characteristics of the topology into account while their “online” scheduler collects network statistics, before optimizing the schedule by moving nodes connected by *hot edges*, i.e. edges that exhibit high data volumes, on the same server. Their evaluations conducted using a synthetic and a real-world dataset show, that online-scheduling results in much lower latency than static or uniform scheduling.

⁴ <http://incubator.apache.org/s4>

3 Problem Statement, Formal Definitions, and System Description

In this section we provide the technical foundations for our study. These include a brief introduction to the data- and processing models employed. Next, we introduce the three concepts of data partitioning, scheduling, and load balancing and how they affect the performance of a distributed system, before presenting a formal problem description. We then show, how the multi-constraint optimization problem of scheduling can be solved using a graph partitioning algorithm, before we, finally, introduce the system we built to test our hypothesis.

3.1 Data- and Processing Model

A linked data stream processing system essentially continuously ingests large volumes of temporally annotated RDF triples and emits the results again as data stream. Such systems usually implement a version of the SPARQL algebra that has been modified for processing dynamic data. In our case, we focus on a subset of those defined in the queries of the *SRBench* benchmark [23]. The processing model considered is a directed graph, where the nodes are algebra operators and data is sent along the edges. Hence, each query can be transformed to a query tree of algebra expressions – the topology of the processing graph.

While the system consumes temporally annotated RDF triples ($\langle s, p, o \rangle [ts]$, where ts denotes the time-stamp), internal operators in the topology consume and emit sets of *variable bindings* when performing the operations associated with the respective operator. These variable bindings comprise a finite number of variable/value pairs $?var/val$, where $?var$ is a variable name and val is an RDF term. Note that source operators (i.e., the input to the topology) consume timestamped RDF triples instead of bindings and output operators may also output RDF triples if so specified by the query.

3.2 Scheduling, Data Partitioning, and Load Balancing

In order to scale the system horizontally (i.e., executing its parts on multiple processing units concurrently) we may replicate parts (or the whole) of the query’s topology and execute clones of the operators in parallel. We refer to these clones as *tasks* or *task instances*. Hence, each operator will be instantiated as a finite number of n tasks (where $n \geq 1$). These task instances, and thus the workload of the system, can then be distributed across several machines in a compute cluster. We refer to the assignment of tasks to machines as *scheduling*. Figure 1 shows an example operator topology and one possible schedule distributing tasks to two servers of a compute cluster. As there are now multiple instances of each operator of the topology the data needs to be *partitioned* in accordance to the operator’s needs. For stateless operators, such as filters or binders, it does not matter which variable bindings they receive for processing. For stateful operators, in contrast, such as aggregators or joins, the system needs to provide some guarantees about

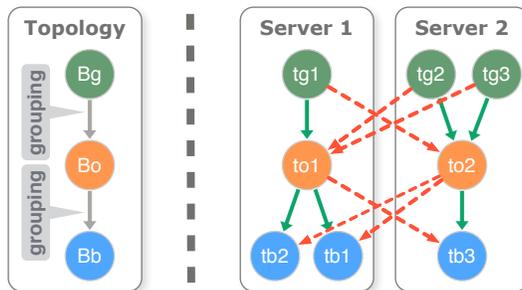


Fig. 1. A topology for an example query with three operators or algebra expressions (left side) and a possible schedule with eight tasks distributed over two servers (right side). Data flows from top to bottom. Green (solid) arrows indicate fast intra-machine communication; red (dashed) arrows indicate costly inter-machine communication.

what data gets delivered to which task instance. To this end, a topology configuration contains *grouping strategies* (or information about the data partitioning function) on the edges between operator nodes (see also Figure 1 on left).

In this study we assume the number of messages sent between the machines as the key performance indicator (KPI). This seems to be a prudent choice, as the network can become a bottleneck of a distributed system that needs to scale horizontally. We acknowledge that our choice has limitations and therefore provide a discussion in Section 5.

Given these definitions *the goal of our approach is to find a schedule (i.e., assignment of tasks to machines) for a given topology that minimizes the total number of data messages transferred over the network, whilst maintaining an even workload distribution across machines in terms of CPU cycles.*

To achieve this goal we partition the data into logical units. We then re-group these using graph-partitioning which provides us with an optimization procedure to minimize the number of messages sent between machines. As a result, we propose the following hypothesis: *Combining data partitioning between tasks with a scheduler that employs graph partitioning to assign the resulting task instances outperforms a uniform distribution of data and tasks to machines.*

Our hypothesis assumes that different distribution strategies significantly influence the number of messages sent between the machines. Most stream processing platforms attempt to uniformly distribute compute loads possibly incurring high network traffic. Approaches like Borealis schedule the processors according to the structure of the query, where every operator is assigned to one machine. This approach has an upper limit in parallelization equal to the number of operators and may incur high network traffic between machines containing active operators. Instead we propose to parallelize the operators and minimize network traffic allowing for more flexibility for distributing the workload. Most importantly, *we propose that the scheduling strategy should optimize the amount of data sent between machines.*

3.3 Formal Problem Description

In principle, a linked data stream processing system can be conceived as a query graph $Q_{SFP} = \langle Op, MC \rangle$, where Op is a finite set of operators op_i (i.e., $Op = \cup_{i=1}^I op_i$) and each op_i executes one or more algebra operations. The flow of information between the operators is established by a set of edges $mc \in MC$ (message channels) that denote a flow of messages (time-stamped variable bindings $\mu[ts]$) between the operators (op_i).

Since we want to enable parallelism and distribution each operator is instantiated in parallel as a finite number of tasks $T_{op_i} = \cup_{j=1}^{J_i} t_{i,j}$, where J_i denotes the degree of parallelism of op_i . We refer to the set of all tasks as

$$T = \cup_{i=1}^I T_{op_i} = \cup_{i=1}^I \cup_{j=1}^{J_i} t_{i,j}$$

Furthermore, each message channel $mc \in MC$ is instantiated via a finite set of channels $c_{ij} \in C$ that connect the tasks $t_i, t_j \in T$. Specifically, connected tasks send messages, i.e., time-stamped variable bindings $\mu[ts]$ to each other.

Thus for each query graph Q_{SFP} there exists a parallelized Task Graph $TG = \langle T, C_T \rangle$, where in addition to the mapping of each task to exactly one operator (as specified above) each channel $c \in C$ maps to exactly one mc and each mc has at least one c to ensure connectivity. Hence, to ensure a correct mapping we require that $\forall op_a, op_b, mc_{op_a, op_b} : \exists t_{a,j}, t_{b,j}, c_{t_{a,j}, t_{b,j}}$, where (i) mc_{op_a, op_b} is a message channel that connects op_a and op_b , and (ii) $c_{t_{a,j}, t_{b,j}}$ connects the corresponding tasks. In addition, we require that $\forall t_{a,j}, t_{b,j}, c_{t_{a,j}, t_{b,j}} : \exists_{\text{exactly one } op_a, op_b, mc_{op_a, op_b}}$ to ensure the one-to- n mapping of operators and message channels to tasks and channels.

Graph Partitioning A partitioning divides a set into pairwise disjoint sets. In our case we want to partition the vertices of a graph $G = (V, E)$ with a finite set of vertices V and a finite set of edges $E \subset V \times V$. A partitioning $P = \{P_1, \dots, P_K\}$ for V separates the set of vertices such that

- it covers the whole set of vertices: $\bigcup_{k=1}^K P_k = V$ and
- the partitions P_k are pairwise disjoint: $\bigcap_{k=1}^K P_k = \emptyset$

In addition, we denote (i) a *partitioning function* by $part : V \rightarrow P$ that assigns every vertex v_1, \dots, v_l the partition $P_k \in P$ it belongs to, (ii) a *cost function* by $cost(P) \in \mathbb{R}$, which denotes some kind of cost associated with the partitioning that is subject to optimization, and (iii) a *load imbalance factor* $stdv(P)$ that ensures that the workload of the tasks is evenly distributed over the machines.

We can easily map our problem of minimizing the number of messages that are sent between machines to a graph partitioning problem with a specific cost function. First, we define the graph to be partitioned as the Task Graph TG . A partitioning of TG maps each task to exactly one machine.

Second, in our case the cost function to minimize is the number of messages sent between machines. We operationalize the cost function for the network traffic as $cost(P) = \sum_{k=1}^K cost(P_k)$. Each $cost(P_k)$ denotes the cost of transmitting

messages, i.e. the bindings, across the network to a task situated in partition P_k . Hence, we increment the cost for $cost(P_k)$ by one, iff a message is being sent from task t_1 to t_2 and the two tasks are *not in the same partition*, i.e. $part(t_1) \neq part(t_2)$ and $t_2 \in P_k$.

Third, when optimizing the costs for the partitions we add the constraint that the partitions shall be balanced with respect to the computational load. We approximate the computation load for a partition by counting the number of messages that are sent over a channel for which task t is the receiving task: $load(P_k) = \sum_{c_{t_a, t_b} \in C} count(m)$ iff $part(t_b) = P_k$, where $m \in M$ is a message sent over channel c_{t_a, t_b} . Note that we count all incoming messages for each task, regardless of whether they had to be transferred over the network or not, as they have to be processed and hence consume computation power in both cases.

In order to make optimal use of the available resources, a balanced load distribution is desirable. The standard deviation $stdv(P)$ of the load for all partitions shall hence not exceed a certain threshold C :

$$C < stdv(P) = \sqrt{\frac{\sum_{p \in P} (\frac{load(p)}{K} - load(P))^2}{K}}$$

All graph partitioning in this paper were computed using the *METIS* algorithms for graph partitioning [10] – a well established graph partitioning package.

3.4 KATTS

In order to test our hypothesis we built a research prototype of a distributed linked data stream processing engine called *KATTS*.⁵ In order to keep the programming overhead minimal, we chose to build our system on top of the *Storm* realtime computation framework.⁶ While our current prototype is built using *Storm* it is important to note, that our findings are not only valid in the context of *Storm*, but for any system that uses partitioned data streams.

A *Storm* application is a graph, consisting of compute nodes that are connected by edges. Edges are configured using a partitioning function or *grouping strategy*. Using the abstractions of *Storm* we implemented a set of stream operators, a configuration environment, and a monitoring suite. Topologies can be specified using XML and will output the sending behavior of the topology: the communication graph. In addition to input operators that read time annotated n-triple files and an output operator, the current set of supported operators contains an aggregation operator (min, max, avg), a filter operator, a bind operator, and a temporal join operator. Every incoming edge of each consuming operator can be configured with a grouping strategy. If no grouping strategy is configured *local or shuffle grouping* will be used.⁷ We always used the *field grouping* strategy, which partitions the data based on the value of a tuple field (i.e., the value

⁵ *KATTS* is a recursive acronym for *Katts is A Triple Torrent Sieve*. The code will be made accessible at <https://github.com/uzh/katts> upon publication of the paper.

⁶ <http://storm-project.net>

⁷ <https://github.com/nathanmarz/storm/wiki/Concepts#stream-groupings>

of a variable). For partitioning *Storm* uses the *hashCode()* method of the field value, which in our case is an object of type *java.lang.String*.⁸ In addition to the configuration parameters of the particular node implementation, each node of the topology can be configured with a value that defines its degree of parallelism, which is the number of task instances that should be created for this operator.

The monitoring suite has two main components: (1) a data collection facility, which records communication behavior, and (2) a runtime monitoring component that keeps track of the number of input sources the system is receiving data from. When all sources have been fully processed, the data aggregation process will be executed and the topology as well as the *Storm* cluster will be halted.

4 Evaluation

In this section we evaluate if our proposed strategy is indeed better in terms of network load whilst maintaining a comparable host load when compared to a baseline strategy of trying to attain a uniform distribution of load. To that end we first present the experimental setup and then present the results.

4.1 Experimental Setup

Datasets We evaluated our system using two example queries that are built around a real world streaming use case: *SRBench* [23], which works with streams of weather measurements, and an open government dataset, which we collected through public sources.

SRBench is an RDF/SPARQL streaming benchmark consisting of weather observations about hurricanes and blizzards in the USA during the years 2001 and 2009 and contains 17 queries on *LinkedSensorData*,⁹ which originated from the *MesoWest* project of the University of Utah.¹⁰ Given that our approach only implements joins, simple aggregates, and triple-pattern matching, we restricted ourselves to Q3, which searches for weather stations observing hurricane-like condition. These are defined as “a sustained wind (for more than 3 hours) of at least 33 m/s or 74 mph.” Without the prefix declaration a C-SPARQL [6] like version of this query would look as follows:¹¹

```
ASK
FROM STREAM <http://www.cwi.nl/SRBench/observations> [RANGE 3h STEP 1h]
WHERE {
  ?observation om-owl:procedure ?sensor ;
              om-owl:observedProperty weather:WindSpeed ;
              om-owl:result [ om-owl:floatValue ?value ] .
}
GROUP BY ?sensor
HAVING ( MIN(?value) >= "74"^^xsd:float )
```

⁸ [http://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode\(\)](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode())

⁹ <http://wiki.knoesis.org/index.php/LinkedSensorData>

¹⁰ <http://mesowest.utah.edu/index.html>

¹¹ We employ a minimum aggregate rather than the average value used in <http://www.w3.org/wiki/SRBench>, as we think the word “sustained” means that the wind speed has to be “at least” 74 miles per hour and not “on average”. We also use a step size of 1 hour instead of 10 minutes.

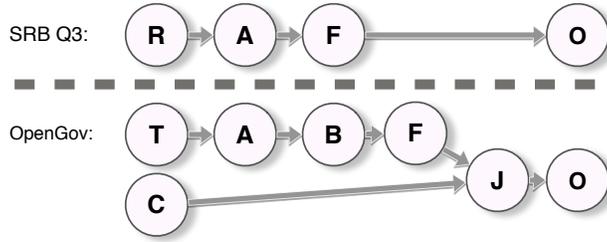


Fig. 2. The topologies for Query 3 of the SRBench and the OpenGov query.

The resulting topology has four processes depicted in Fig.2: First, the reader node (R) reads the incoming stream and scans it for the triple patterns contained in the where-clause. Second, the matched bindings are then sent to the aggregator node (A), which creates the minimum aggregate over the temporal window of three hours and a step size of one hour. Third, the output of the aggregator is then sent to the filter node (F), which filters all occurrences that are smaller than 74 mph and sends all remaining instances to the output node (O). Finally, the output node writes the occurrences into a file on disk.

OpenGov Dataset: To complement the regular setup of *SRBench*, which consists mostly of weather station measurements, we gathered a second data set, which combines data on public spending in the US with stock ticker data.¹² We devised a query that would highlight (publicly traded) companies, that double their stock price within 20 days and are/were awarded a government contract in the same time-frame. This query requires the system to scan two sources, aggregate/filter values, and finally join certain events that may have a causal relation to each using a temporal condition. The C-SPARQL representation of the query, for example, looks as follows:

```
REGISTER QUERY PublicSpendingStock AS
SELECT { ?company_name ?agency_name ?contract_id ?min_price ?max_price ?factor }
FROM STREAM <wrds.crsp/ticker.trdf> [RANGE 20 DAY STEP 1 DAY]
FROM STREAM <usaspending.org/contracts.trdf> [RANGE 20 DAY STEP 1 DAY]
WHERE { GRAPH <wrds.crsp/ticker.trdf> {
    ?ticker_id wc:PRC ?ticker_price ;
              wc:COMNAM ?company_name ;
              wc:TICKER ?ticker_symbol .
  } UNION GRAPH <usaspending.org/contracts.trdf> {
    ?contract_id us:agencyid ?agency_name ;
                us:obligatedamount ?contract_amount ;
                us:vendorname ?company_name .
  }
}
AGGREGATE { (?min_price, MIN, {?ticker_price}) }
AGGREGATE { (?max_price, MAX, {?ticker_price}) }
BIND (?max_price / ?min_price AS ?factor)
FILTER(?factor > 2)
```

The resulting topology (Fig.2) first aggregates (A) the ticker-sourced (T) data to compute the minimum and maximum over a time window of 20 days. It computes the ratio between these numbers (B), and then filters those solutions where that ratio is smaller than or equal to two (F). The remaining company tickers are then joined (J¹³) with the ones that were awarded government contracts (C). The joined tuples are then sent to the output node (O).

¹² <http://www.usaspending.gov>, <https://wrds-web.wharton.upenn.edu/wrds>

¹³ We use a hash join with eviction rules for the temporal constraints.

Evaluation Criteria In accordance with the Properties-Challenges-KPIs-Stress-tests (PCKS) paradigm for benchmarking SFP systems [18] we tested the performance of our Distributed Flow Processing System by choosing the number of inter-machines network messages as a key performance indicator (KPI). As a secondary performance indicator (SPI) we chose the uniformity of load distribution as measured by number of messages received per machine.

Procedure We used the following procedure to measure the performance of our approach. First, we took each dataset and partitioned it to 12 files (as we had 12 machines at our disposal). The two queries were compiled into the topologies described above and instantiated to allow 48 tasks for each node that is neither a reader nor an output node.¹⁴ We then recorded the number of messages that were sent between tasks at runtime.

Second, to test our hypothesis we needed to partition the resulting communication graph based on the network load of each channel. Since the channel loads are not known before running the query we chose two experimental scenarios. In the first scenario we assume an *oracle* optimizer that would know the number of messages that would flow along every channel. This scenario allows to establish a hypothetical upper bound of quality that our method could attain, if it were to have an oracle. In a second scenario we assumed a *learning* optimizer that first observes channel statistics for a brief period of time and then partitions the graph accordingly. To that end we sliced the *SRBench* data into daily and the *OpenGov* data into monthly slices. We then measured the performance of our approach based on learning during the preceding one to three time-slices, essentially providing a adaptively learning system.

Third, to partition the graph we employed *METIS* [10]. We used the *gpmetis* in its standard configuration, which creates partitions of equal size, and only changed the *-objtype* parameter to instruct *METIS* to optimize for total communication volume when partitioning, rather than minimizing on total edgcut.¹⁵

4.2 Results

The Suitability of Graph Partitioning for Scheduling The results of our evaluation are shown in Figure 3, which plots the number of network messages divided by the number of total messages as a measure for the optimality of the distribution. As the figure shows on the left, the *SRBench* data can be optimally partitioned by the id of the reporting weather station even when using only the data of the immediately preceding time slice (Prev.1). All further computation can be managed on a local machine, as no further joins are necessary. This clearly indicates that some queries can be trivially distributed when a good data partition is either known or can be learned.

On the right we find the results for the *OpenGov* dataset. This evaluation is not quite as clear-cut, as the join operation requires a significant redistribution of messages. The results here are quite interesting. First, we find that our approach

¹⁴ As we ran our experiments on machines with more than 12 cores, we were able to achieve better capacity utilization by using more than 12 tasks per node.

¹⁵ We used v5.0.2 with default partitioning (kway) and default load imbalance of 1.03.

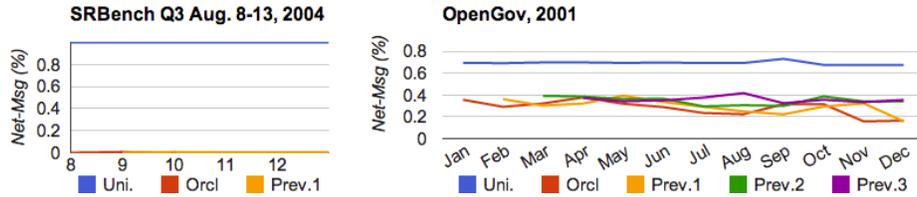


Fig. 3. Percentage of messages sent over the network for the uniform distribution and the graph partitioned setup, using either the test data itself (oracle) or data from the previous one to three time-slices as input for the graph partitioning algorithm.

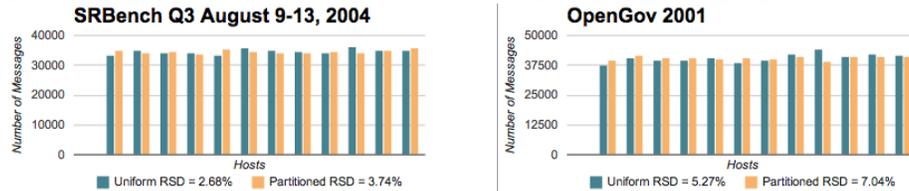


Fig. 4. Average computation load distribution for all time-slices of each dataset. RSD = Relative Standard Deviation

clearly outperforms the uniform distribution strategy by a factor of two to three. Second, it is interesting to observe that even longer learning periods, using two (Prev.2) and even three previous time slices (Prev.3), do not necessarily improve the overall performance - maybe due to over-fitting or concept drift [19].

For the sake of brevity we only show data for three time-slices of each evaluation in Table 1: on the left side again the results for the embarrassingly parallel *SRBench* query, which shows a reduction in network usage by over 99%. The right side of the table is more interesting as it exhibits the gain of our approach in a non-trivial case. Even for the *OpenGov* query, workload distribution using a graph partitioning approach yields savings of network bandwidth of over 40%.

Balancing Computation Load Next to keeping the bandwidth usage to a minimum, a distributed system must also make good use of the available computational power. For this reason we analyzed how many messages were processed by all tasks on each host for the two queries. Figure 4 shows the results of this

| <i>SRBench</i> Q3 August, 2004 | | | | <i>OpenGov</i> , 2001 | | | |
|--------------------------------|---------|--------|--------|-----------------------|---------|--------|--------|
| Slice | Uniform | Oracle | Prev.1 | Slice | Uniform | Oracle | Prev.1 |
| Aug 9 | 1 | 0.7% | 0.0% | Feb | 69,5% | 29.0% | 36.0% |
| Aug 10 | 1 | 0.0% | 0.7% | Mar | 69,8% | 32.1% | 30.1% |
| Aug 11 | 1 | 0.0% | 0.0% | Apr | 69,8% | 37.7% | 32.1% |

Table 1. Percentage of messages sent over the network for the uniform distribution, the partitioning based on the test data itself (oracle), and the preceding time slice (“Prev.1” in Table) as input for the graph partitioning algorithm; three time slices each.

evaluation: The load distribution resulting from the graph partitioned task assignment only differs slightly from the one found by uniform task distribution (average relative standard deviation [RSD] *OpenGov*: 7.04% for partitioning vs. 5.27% for uniform baseline; *SRBench*: 3.74% for partitioning vs. 2.68% for uniform baseline).

The Influence of Data Partitioning The results above are very encouraging. One of the major limitations of our measurements, however, is that we assumed that the data came partitioned into meaningful groups. Whilst this assumption is often true in practice (the input from weather stations comes as grouped messages from one station, data about one stock usually arrives from one source, etc.). But in some worst-case scenarios the data might be mixed (even if a total random intermixing is unlikely). To investigate the robustness of our procedure against this assumption we ran our approach under two different partitioning regimes: first, we made sure to partition the data along a different hash function than chosen by our system (which relies on the *Storm* hash partitioning) and second, we ensured employing the same partitioning. The results of this sensitivity analysis are shown in Figure 5 for the *SRBench* query, which graphs a Sankey chart of the inter-task communication under both conditions, where the width of the lines corresponds to the number of messages. As the figure clearly shows the mixed hashing setting requires to reshuffle all data from the readers to the processing nodes, while the equally partitioning setting provides a clean stream setting. As a consequence, we can expect that badly pre-partitioned data would not exhibit as good results as the ones we exhibited above.

5 Discussion and Limitations of the Results

The results shown in the section above show that using a graph partitioning algorithm to schedule tasks instances on machines does indeed reduce the messages sent over the network whilst only having a slightly less even load distribution.

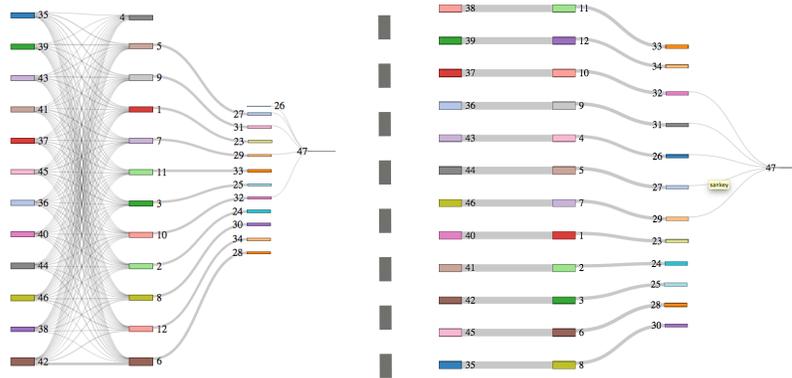


Fig. 5. Two communication graphs (data flows from left to right)
 Left: Input partitioned using different hash function than the one used by Storm.
 Right: Input partitioned using identical hash function as the one used by Storm.

The first part of the finding could be seen as almost tautological: it could be understood as showing that graph-partitioning using a well-established algorithm is better than a partitioning that ignores network traffic but “only” focuses on load distribution. We believe, however, that there are subtle considerations that are less than obvious.

First, the *critical element is to realize that the operators can be parallelized with an adequate data partitioning approach* not to “just” use graph partitioning. It is the interplay of the two partitionings that enables the graph partitioning to find a good schedule: inadequate data partitioning can lead to highly suboptimal schedules as the results about the influence of data partitioning show.

Second, the principle of *finding the smallest possible partition given the desired degree of parallelism* (see also Section 3.2) seems important. How important needs to be investigated. Whilst the idea seems simple its details are intricate and required careful analysis—a task that we will have to continue in the future by further exploring the interactions between data and graph partitioning and devise an automated model for optimizing it.

One somewhat surprising outcome of our analysis is that the overall efficiency of the system heavily depends not only on the *consistent use* of the same partitioning function, but also on the *compatibility of the values* over which the data is being partitioned. Using incompatible data partitioning functions can result in very poor performance as seen in section 4.2. If a topology contains operators that partition over incompatible fields such as in the OpenGov query, graph partitioning is still useful, but much less effective as when working with compatible fields. It is this observation which contains an interesting insight: *linked data stream processors should work with graph fragments rather than triples*. “Naturally” occurring graph fragments often contain interdependent graph elements. If one pulls these fragments apart due to some partitioning function one might have to gather them in a later join. Hence, it seems prudent to favor an approach that leaves these fragments together if a later join is foreseeable.

Our current analysis is based on some *underlying assumptions*. First, we assumed that some *network statistics are available* at the onset. Whilst this may not always be given, our findings show that even a small amount of statistics seem to produce adequate schedules. Hence, it seems straightforward to start with an uniform distribution and then apply an incremental graph partitioning approach [14] improving the schedule during run-time.

Second, we assumed that the *processing load (both CPU and RAM) is proportional to the number of messages received* (i.e., constant operator complexity). Whilst this assumption is definitely true for some operators (e.g., computing the average) others may require more computational effort. We intend to address this issue in future work.

Third, we assumed that the *query topology was given*. Obviously, queries could be translated to various topologies; each of which would require its own schedule. Hence, it would make sense to combine our approach with a query optimizer – a task beyond the scope of this paper.

Also, our current evaluation has some limitations. First, it is limited to two datasets and queries. Whilst the queries seem representative of many settings we have seen we intend to significantly extend our evaluation in the future in terms of number of datasets and queries. Second, all our evaluations were run on a cluster with 12 machines, 1GB ethernet, and 24 cores each. Obviously, we will have to extend our evaluation to investigate the interactions between number of machines and cores available and the degree of parallelism “granted.” Third, we will have to run throughput-analyses in real-world setups in addition to our current network analysis adding number of messages ingested per second as KPI.

6 Conclusion and Outlook

In this study we investigated whether and how scheduling the tasks of Distributed Semantic Flow Processing (DSFP) systems benefits from applying graph partitioning. We implemented our approach on the Katts DSFP engine and evaluated it using a query of the *SRBench* benchmark and a usecase inspired by the open government movement with regards to network load. The results show that using a graph partitioning algorithm to schedule task instances on machines does indeed reduce the number of messages sent over the network. We also found that this only leads to a slightly less even load distribution.

The critical element for optimizing the scheduling using graph partitioning is an adequate data partitioning for parallelizing the operators. Future work will investigate whether the principle of finding the smallest possible data partition given the desired degree of parallelism is as important as our experiments indicate.

Our study’s most important shortcomings are its limitation to two datasets and queries and the fixed setup of the distributed system. For the first we intend to systematically extend our evaluation in the future in terms of number of datasets and queries. For the latter, is it the interactions between number of machines and cores available and the degree of parallelism that require further research. Especially the impact of such interactions on throughput in terms of messages ingested per second is of interest here.

We are confident that our findings help making DSFP systems more scalable and ultimately enable reactive systems that are capable of processing billions of triples or graph fragments per second with a negligible delay. It is our firm belief that the key to addressing these challenges needs to and will have to be revealed from the data itself.

Acknowledgements We would like to thank Thomas Hunziker, who wrote the first prototype of the *KATTS* system during his master’s thesis in our group.

References

1. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., Rasin, A., Et Al.: Aurora: a data stream management system. In: Proc. of the 2003 ACM SIGMOD. pp. 666–666 (2003)

2. Abadi, D.J., Ahmad, Y., Balazinska, M., Hwang, J.h., Lindner, W., Maskey, A.S., Rasin, A., Ryzkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proc. CIDR2005. pp. 277–289 (2005),
3. Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Park, Y., Venkatramani, C.: Spc: A distributed, scalable platform for data mining. In: Proc. Workshop on Data Mining Standards, Services and Platforms, DM-SSP (2006)
4. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW2011. pp. 635–644 (2011)
5. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: DEBS2013 (2013),
6. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A Continuous Query Language for RDF Data Streams. *Int. J. of Sem. Comp.* 4(1), 3–25 (2010)
7. Calbimonte, J.p., Corcho, O., Gray, A.J.G.: Enabling Ontology-based Access to Streaming Data Sources. In: Proc. ISWC 2010 (2010)
8. Cugola, G., Margara, A.: Processing flows of information. *ACM Computing Surveys* 44(3), 1–62 (Jun 2012),
9. Hoeksema, J., Kotoulas, S.: High-performance Distributed Stream Reasoning using S4. In: First International Workshop on Ordering and Reasoning (2011)
10. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. on Scientific Comp.* 20(1), 359–392 (Jan 1998),
11. Komazec, S., Cerri, D.: Sparkwave: Continuous Schema-Enhanced Pattern Matching over RDF Data Streams. In: DEBS 2012 (2012)
12. Lajos, J.F., Toth, G., Racz, R., Panczel, J., Gergely, T., Beszedes, A.: Survey on Complex Event Processing and Predictive Analytics. Tech. rep., Citeseer (2010),
13. 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: Proc. ISWC 2011. vol. 7031, pp. 370–388 (2011)
14. Ou, C.W., Ranka, S.: Parallel incremental graph partitioning. *Parallel and Distributed Systems, IEEE Transactions on* 8(8), 884–896 (1997)
15. Owens, T.: Survey of event processing. Tech. Rep. December, Air Force Research Laboratory Public Affairs Office (2007),
16. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-Aware Operator Placement for Stream-Processing Systems. In: Proc. ICDE2006 (2006)
17. Rinne, M., Nuutila, E., Seppo, T.: INSTANS : High-Performance Event Processing with Standard RDF and SPARQL. In: ISWC 2012 Post. & Demos. pp. 6–9 (2012)
18. Scharrenbach, T., Urbani, J., Margara, A., della Valle, E., Bernstein, A.: Seven Commandments for Benchmarking Semantic Flow Processing Systems. In: ESWC 2013 (2013)
19. Vorburger, P., Bernstein, A.: Entropy-based Concept Shift Detection. In: Proc. ICDM2006. pp. 1113–1118 (2006)
20. White, T.: Hadoop: The definitive guide. O’Reilly Media, Inc., 3 edn. (2012),
21. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L., Fleischer, L.: SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In: Proc. Middleware2008 (2008)
22. Xia, C., Towsley, D., Zhang, C.: Distributed resource management and admission control of stream processing systems with max utility. In: Proc. ICDCS2007 (2007)
23. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.p.: SRBench : A Streaming RDF / SPARQL Benchmark. In: Proc. ISWC 2012 (2012)