Locality-Aware Routing in Stateful Streaming Applications

Matthieu Caneill, Ahmed El Rheddane, Vincent Leroy, Noël de Palma

Locality-Aware Routing in Stateful Streaming Applications

Matthieu Caneill
caneill@imag.fr

Ahmed El Rheddane
elrhedda@imag.fr

Vincent Leroy
leroy@imag.fr

Noël De Palma
depalma@imag.fr

Univ. Grenoble Alpes, LIG, CNRS
F-38000, Grenoble, France

ABSTRACT

Distributed stream processing engines continuously execute series of operators on data streams. Horizontal scaling is achieved by deploying multiple instances of each operator in order to process data tuples in parallel. As the application is distributed on an increasingly high number of servers, the likelihood that the stream is sent to a different server for each operator increases. This is particularly important in the case of stateful applications that rely on keys to deterministically route messages to a specific instance of an operator. Since network is a bottleneck for many stream applications, this behavior significantly degrades their performance.

Our objective is to improve stream locality for stateful stream processing applications. We propose to analyse traces of the application to uncover correlations between the keys used in successive routing operations. By assigning correlated keys to instances hosted on the same server, we significantly reduce network consumption and increase performance while preserving load balance. Furthermore, this approach is executed online, so that the assignment can automatically adapt to changes in the characteristics of the data. Data migration is handled seamlessly with each routing configuration update.

We implemented and evaluated our protocol using Apache Storm, with a real workload consisting of geo-tagged Flickr pictures as well as Twitter publications. Our results show a significant improvement in throughput.

CCS Concepts

•**Information systems** → *Data stream mining;*

Keywords

stream processing, network locality, routing, optimization

1. INTRODUCTION

Stream processing engines such as Apache Storm [22] and Apache Spark Streaming [24] have become extremely popular for processing huge volumes of data with low latency. Streams of data are produced continuously in a variety of context, such as IoT applications, software logs, and human activities. Performing an online analysis of these data streams provides real-time insights about data. For instance, the Twitter infrastructure processes up to 150,000 tweets per second, and maintains a list of trending hashtags. This cannot be done using batch jobs, as the processing delay would make the result irrelevant by the time they are produced. Stream processing solves this problem by continuously analysing the new tweets in memory, and updating results within milliseconds.

A stream application consists of a directed acyclic graph in which vertices are operators, and edges represent data streams between operators. To scale horizontally, each operator is deployed as several instances distributed over multiple servers. Stateful applications maintain statistics on different topics, and use *fields grouping* to ensure that data tuples related to a given *key* are always sent to the same instance of an operator. This assignment usually relies on hash functions to obtain a random but deterministic routing. As the application is deployed on a higher number of servers, the likelihood that the recipient operator instance is located on a different server increases. This increases the network consumption of the application, and significantly degrades its performance.

In this work, we propose to build optimized routing tables to improve network locality in streaming applications. We analyse the correlation between keys used in successive fields grouping, and explicitly map correlated keys to operator instances executed on the same server. Hence, data tuples containing these keys can be passed from one operator to the next through an address in memory, instead of copying the data over the network. This has the advantage of being faster, and avoids the saturation of the network infrastructure. Hash functions suffer from skews in data distribution, as the most frequent keys cause load imbalance on operator instances responsible for processing them. As we collect statistics on the distribution of keys, routing tables can also be used to ensure that the load remains even, which further improves the throughput of the application.

Data streams fluctuate over time, and association between keys can vary significantly. Consequently, we opt for an online approach to optimize routing and reconfigure key routing without disrupting the application. To this end, we add an instrumentation tool to stateful operators that gather statistics on the frequency of key pairs. A *coordinator* collects these statistics from all operators, and executes a graph

Middleware'16, December 12 - 16, 2016, Trento, Italy

ISBN 978-1-4503-4300-8/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2988336.2988340

partitioning algorithm to divide keys between servers while balancing the load. Finally, an online data migration protocol ensures that the state of reassigned keys is migrated between operator instances without data loss. We evaluate our approach on Apache Storm. Our workloads consist of synthetic datasets with varying degrees of key correlations, and two real datasets from Twitter and Flickr. The results show a significant improvement in throughput.

The rest of this paper is organized as as follows: Section 2 provides background information on streaming applications. Section 3 describes our approach for optimizing data routing. We present the evaluation in Section 4. We review the related work in Section 5, and conclude in Section 6.

2. BACKGROUND

We present in this section some general stream processing concepts, before focusing on stream routing between operators.

2.1 Stream processing

Application model.

Stream processing was developed to continuously execute operators on potentially unbounded streams of data tuples. Apache Storm [23], Flink [1], S4 [18], Samza [2], and Twitter Heron [14], are examples of popular stream processing engines. Following the dataflow programming paradigm, a stream processing application can be described as a Directed Acyclic Graph (DAG). Vertices represent processing operators (POs) that consume and produce data streams, which flow along edges. A source constitutes the entry point of the DAG, and streams data tuples, such as posted tweets or uploaded pictures, to POs. Stream processing implements a share-nothing policy, so operators can be executed in parallel as they manipulate different tuples of data. Figure 1 represents a simple wordcount application for streams of sentences. Data enters the DAG at the first PO A and flows from the left to the right. A takes as input sentences and extracts words, thus producing a stream of words. These words then reach B which, for each input word, writes the same word in lower-case format to its output stream. Finally, C counts the frequency of each word in its input stream. POs A and B are stateless, as they do not update any internal state when processing data, while C is stateful as it maintains frequency counts (stateful POs and POIs are represented with double circles in the remainder of the paper). In a general case, each PO can input and output multiple streams of data. In this paper, for the sake of simplicity, we present applications consisting of chains of POs, in which each PO has a single input and output stream. The results presented remain, however, valid for more complex DAGs. We also focus on streaming frameworks which implement long-running tasks (such as Apache Storm), as opposed to the micro-batch processing model (as in Apache Spark).

Scalability.

This approach to stream processing scales horizontally by placing different POs on different machines. Moreover, each PO of a DAG can be replicated into different processing operator instances (POIs) to increase its throughput. Figure 2 presents a possible deployment of the wordcount application. PO A is replicated twice, with POIs A_1 and A_2 executing

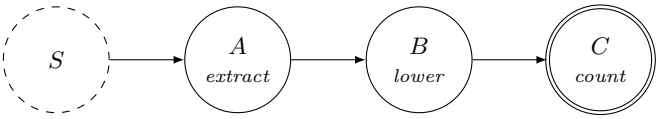


Figure 1: A simple wordcount stream application. S sends sentences, operator A extract words, B converts them to lowercase, and C counts the frequency of each word.

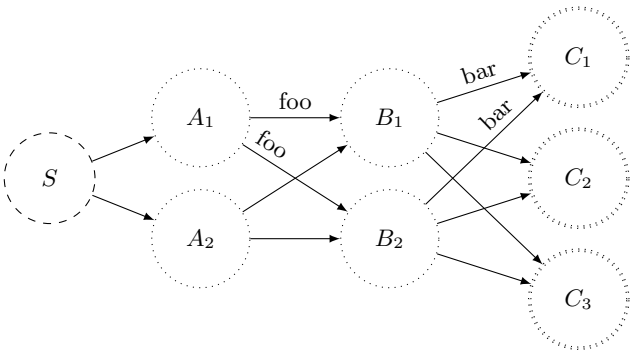


Figure 2: Three components of a DAG having respectively 2, 2 and 3 instances each. A and B are stateless while C is stateful.

the same code in parallel. In the remainder of this paper, X_i refers to an instance of PO X executed on server number i. In the case of a stateless PO such as A and B, this replication is trivial: since no state is maintained, it does not matter which instance of the PO processes a given tuple of data. Hence, data streams can be split arbitrarily between POIs of the same PO. However, replicating stateful POs such as C is more complex. If a word w appearing multiple times in the input sentences is processed each time by either C_1, C_2 or C_3, two problems appear. The first one relates to scalability. Each of the 3 POIs of C maintains a state related to the frequency of w, so the total memory consumption of C increases linearly with the number of replicas. The second problem is that no single POI of C has the correct information regarding the frequency of w. It is impossible for a POI of C to detect when the frequency of w reaches a given threshold, in order to trigger an action for instance. Thus, it is important to ensure that all occurrences of w are routed to the same instance of C. While replicating processing operators is key to scale stream processing, it must be used in combination with an appropriate stream routing policy. We detail stream routing policies in Section 2.2.

2.2 Stream routing policies

The specification of the application DAG indicates which PO is the recipient of another PO's output stream. When several POIs are deployed for a given PO it is important to also select which particular POI receives each tuple of data from a stream: this is the role of the routing policy. Each edge of the DAG is labeled with a choice of routing policy indicating how the output stream of a PO is split between the POIs of the recipient PO. When a POI X_i emits data for a POI Y_i, both POIs are executed by the server i, so passing the data is extremely fast. Only an address in memory is transmitted from a thread to another. However, when X_i communicates with Y_j with $i \neq j$, the

two POIs are deployed on different servers, which can be on different racks, or even different locations. The tuple goes through the network, which is less efficient and can constitute a bottleneck for the system. We now review the 3 main stream routing policies.

Shuffle grouping.

Data tuples are sent to a POI of the next PO with a round robin fashion. This ensures a perfect load balancing between the POIs. This can however only be used when routing to a stateless PO, as described in Section 2.1. POI co-location is not taken into account, meaning that a message can be routed to another server even if there is an instance of the destination PO on the current server. This incurs a significant network overhead. This is particularly inefficient when successively executing two stateless POs, such as A and B in the example of Figure 2. If B is deployed over 5 POIs, then 80%, i.e. 4 tuples out of 5, of the data from A to B goes through the network. This proportion only grows as the system is deployed on larger clusters. To solve this issue, A and B could be combined in a new PO AB executing both operators consecutively. This would avoid network communication, but drastically limits the modularity and reusability of the POs.

Local-or-shuffle grouping.

This grouping is an optimized version of the shuffle grouping: when a POI of the recipient PO is located on the same server, it is selected. This policy opportunistically avoids needless network communication, and mimics the existence of the AB PO mentioned above. The guarantees in terms of load balancing are slightly weaker, but for most applications, if the input of a A is balanced then its output, and thus the input of B, remains balanced without the need for a round robin assignment. Similarly to shuffle grouping, this grouping is not appropriate for stateful POs.

Fields grouping.

This policy is used when routing to a stateful PO. The developer selects fields of data tuples as a key to determine the recipient POI, similarly to the reduce function in MapReduce [10]. Hence, all tuples having the same key are sent to the same POI. In the example of Figure 2 each word is used as a key when routing from B to C. The default implementation of fields grouping uses a hash function on the key to determine the recipient POI, but it is also possible to define and maintain routing tables to explicitly assign keys to POIs. Fields grouping is necessary to ensure the consistency of stateful POs. However, it can lead to load balancing issues when the distribution of keys is skewed.

3. LOCALITY-AWARE ROUTING

In this section, we first state the problem, before providing our solution. In order to do so, we describe how we identify the correlation between keys by collecting statistics from the different POIs. We then focus on the generation of the routing table, reduced to a graph clustering problem. Finally, we describe our protocol that allows the routing tables of POIs to be changed dynamically.

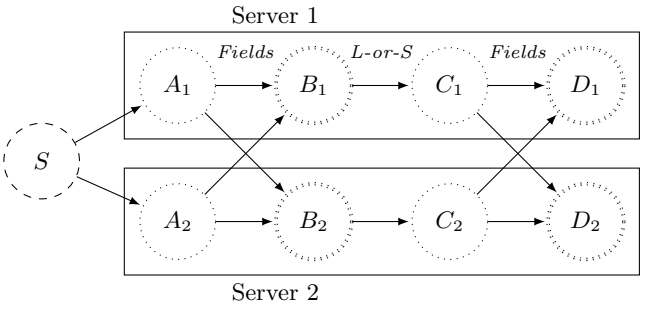


Figure 3: Deployment of a stateful streaming application, with fields grouping linking POs A and B, local-or-shuffle grouping linking POs B and C, and fields grouping linking POs C and D.

3.1 Problem statement

In this paper, we tackle the problem of optimizing stream processing applications. In Section 2.2, we explain that network communication constitutes a bottleneck in stream processing. Such bottleneck can easily be avoided when routing to stateless POs through the use of local-or-shuffle grouping. Hence, we argue that the main limitation comes from stateful POs that require fields grouping to ensure consistency. Our objective is to maximize the locality of streams routed using fields grouping.

Figure 3 shows a possible deployment of a stream processing application. Each of the 4 POs has two POIs, deployed on two different servers. A and C are stateless, while B and D are stateful, and maintain a state related to the keys of the data tuples they receive, using a hash map for example. Stream routing policies are indicated on the edges between POIs. Routing to C is local to each server, but routing to B and D incurs network traffic between servers. Our goal is to minimize the amount of traffic spanning across servers, i.e. the traffic along (A_1, B_2), (A_2, B_1), (C_1, D_2), (C_2, D_1). More generally, given a stream processing application with POs replicated across different servers, for any two POs X and Y connected through fields grouping, our goal is to minimize:

$$\sum_{i \neq j} \mathit{traffic}(X_i, Y_j)$$

where the *traffic* function indicates the number of data tuples sent from one POI to another. A trivial solution to this problem is to process all data on a single server, which negates the benefits of deploying the application on an additional server. Hence, we add the constraint that the load of the application should remain balanced between POIs: the number of data tuples received by a POI should not be higher than α times the average number of tuples received by POIs of the same PO, where $\alpha \geq 1$ is the imbalance bound.

In this work, we assume that the deployment of POIs on servers is static. Our contribution is a protocol that generates optimized routing tables for fields grouping. We propose to detect at run time the correlations between keys used for consecutive field groupings in order to ensure that those keys are handled by POIs located on the same server. For instance, if the POI B_1, after receiving a tuple with key k, frequently leads to sending a tuple with key k' to PO D,

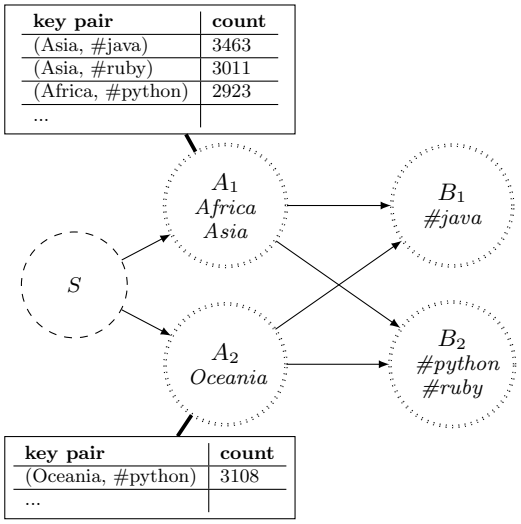


Figure 4: PO instrumentation: every instance counts the key pairs it receives and sends, and keeps the most frequent pairs in memory.

then we should make sure that D_1 is the POI responsible for k'. As streaming applications are executed on unbounded streams, the characteristics of data may change over time. The routing optimization protocol detects these changes and generates new appropriate routing tables. In addition, as a key is assigned to a different POI, a data migration protocol ensures that the state of the application is preserved.

3.2 Identifying correlations

Our goal is to leverage the correlation between keys used in consecutive fields grouping to avoid routing streams over the network. The first step consists in detecting candidate pairs of keys that show potential for optimization. Let X and Y be two consecutive stateful POs with instances over a set of n servers. Given two keys k and k' used when routing to X and Y respectively, the probability of the POIs processing a tuple containing k and k' being located on the same server is $1/n$ when using hash functions. Hence, by ensuring that k and k' route to the same server, the expected number of network messages avoided is $f(k,k').(n-1)/n$ where $f(k,k')$ is the number of data tuples containing k and k'. This shows that *(i)* routing optimization becomes increasingly important as the number of servers increases, and *(ii)* frequent pairs lead to the highest gains. Indeed, even if two keys always appear simultaneously, and thus have a very strong correlation, the gains are negligible if they are not frequent. Conversely, a loose correlation can lead to significant gains if the keys are extremely frequent.

In the remainder of this section, we use the following application as a running example: geolocated short messages containing hashtags (tweets) are analysed to generate statistics about topics trending in geographical regions [7]. The application contains two stateful POs, and routes first using the region, and then using the hashtag. If the pair $(Asia, \#scala)$ appears more often than the pair $(Asia, \#clojure)$, it means that people in Asia tweet more often about Scala than Clojure. In this case, it is more important to co-locate on the same server the POIs dealing with the keys $Asia$ and $\#scala$ than $Asia$ and $\#clojure$.

Offline analysis.

In cases where the workload is stable, correlations between keys are assumed to remain constant over time. Consequently, it is possible to perform an offline analysis on a large sample of the data and to accurately compute the frequency of all key pairs. This information can then be used to compute optimized routing tables that can be used for long periods of time without the need for an update.

Online analysis.

Data streams often fluctuate over time, particularly when they are generated by human activity. For example, $\#breakfast$ is associated to *America* and *Europe* at different moments of the day. In addition to diurnal and seasonal patterns, flash events can occur, generating temporary correlations between keys. It is necessary to detect these correlations at run time to perform an online optimization of stream routing without interrupting the execution of the application. For this purpose, we add an instrumentation tool to stateful POs. For each passing message, a POI extracts the input key, which was used to route the data tuple to this instance, and the output key, which decides towards which POI the message is routed next. Pairs of keys are stored in memory along with their frequency, as depicted in Figure 4. Computing the frequency of pairs of keys online is a challenging problem, as most of the resources, such as CPU and memory, should be dedicated to the application, and not collecting statistics. To this end, we rely on the SpaceSaving algorithm [15]. Using a bounded amount of memory, it maintains an approximated list of the n most frequent pairs of keys. This limitation on the collection of statistics is, fortunately, not problematic for most large-scale datasets. Indeed, many real datasets follow a Zipfian distribution [4], with few very frequent keys, and many rare keys. Identifying the pairs containing the most frequent keys captures most of the potential for optimization, so the loss compared to an exact offline approach is limited. Whenever the routing of keys is updated, the statistics are reinitialized to only take into account recent data and detect new trends.

3.3 Generating routing tables

In addition to the streaming application, we execute a *Manager*, responsible for analysing the statistics collected by the POIs and coordinating the deployment of an optimized routing configuration. For each pair of consecutive stateful POs X and Y, the manager periodically queries all POIs of X to obtain their statistics on the frequency of associations between keys. These statistics can be represented as a bipartite graph connecting keys of X to keys of Y. Each key is a vertex weighted by its frequency, and an edge between keys is weighted by the frequency of their co-occurrence. Figure 5 shows the bipartite graph, as it would be constructed with the data in Figure 4. The pairs observed the most frequently are represented by bold edges.

Creating the best assignment of keys to servers reduces to a graph partitioning problem. We want to partition the bipartite graph such that pairs of keys that appear together frequently are in the same group. In other words, we try to miminize the added weights of the cut edges. That way, we can co-locate on the same servers the instances which deal with correlated keys and decrease the network utilization. In Figure 5, assuming the application is deployed on $n = 2$ servers, $Asia$, $\#java$ and $\#ruby$ should be put on the first

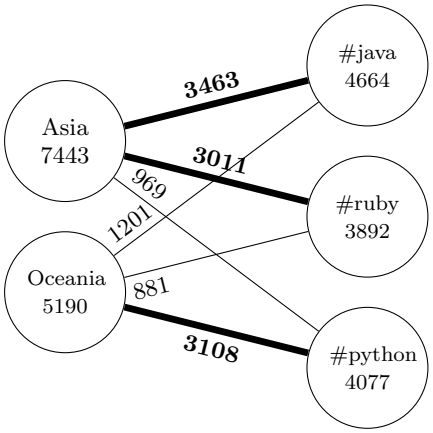


Figure 5: A bipartite graph of the key pairs, showing different weights for the vertices and edges, i.e., pairs.

one, while *Oceania* and *#python* on the second one. Load balancing between servers is handled by balancing the sum of vertex weights in each partition. In practice, we rely on the Metis partitioning library [12]. We construct the bipartite key graph and provide it along with a balance constraint to Metis, which in turn returns the partitioned graph. At this point, keys are assigned to servers, so we can generate routing tables mapping these keys to the POIs hosted on each server.

Each POI preceeding a stateful PO uses the optimized routing table to route a key to a POI. When a key is not present in the routing table, it falls back to the standard hash-based routing policy.

3.4 Online reconfiguration

In the case of an offline analysis, optimized routing tables can be loaded at the start of the application. However, the online approach requires the manager to send the updated routing tables to POIs, while not losing any data tuple of the stream in the process.

The main difficulty when dealing with hot reconfiguration is state migration. Every stateful POI holds the state of the keys to which it is associated. When a key is assigned to a different POI in the updated routing tables, its corresponding state needs to be transferred between POIs. Moreover, after POIs migrate the state of their previous keys, they should no longer receive any message related to this key. This means that preceding POs in the DAG must have proceeded to their reconfiguration first, and route messages according to the new routing tables. To this purpose, we implemented a protocol which orchestrates a progressive reconfiguration following the PO order specified by the DAG. This protocol is shown in Figure 6 and described more formally using pseudo-code in Algorithm 1.

The reconfiguration protocol is executed by the manager M. The manager first asks every running POI to send the collected statistics ①. Upon receiving them all ②, it builds the bipartite graph of the key pairs (see Figure 5), partitions this graph with Metis, and computes the new routing tables. It sends these tables to the respective POIs ③, and waits for all acknowledgements ④. It then enter the propagates phase, and tell the instances of the first PO to proceed to the reconfiguration ⑤. The two instances update their

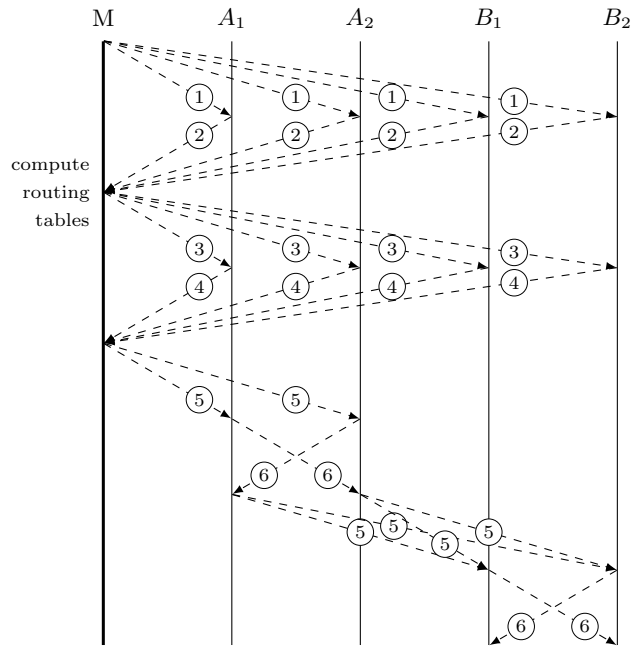


Figure 6: Reconfiguration protocol, forwarding routing tables and key states between POIs. ① Get statistics. ② Send statistics. ③ Send reconfiguration. ④ Send ACK. ⑤ Propagate. ⑥ Exchange keys.

routing table and exchange the state of the keys whose assignment has changed ⑥. After this operation, they forward the propagation instruction to the instances of the second PO ⑤, which in turn update their routing tables and exchange their states if necessary ⑥.

The computational cost of this reconfiguration is linear in time with respects to the number of POs, and linear in space with respects to the number of key pairs which are counted.

The reconfiguration message sent by the manager to every instance is a data structure containing:

- *reconfiguration_router*: The new routing table, associating keys to POIs.
- *reconfiguration_send*: The list of keys whose state must be transferred along with their respective recipients.
- *reconfiguration_receive*: The list of keys whose states are expected to be received from other instances of the same PO.

The data stream is not suspended during reconfiguration, so it is possible that a POI receives a tuple associated to a key while it has not yet received the state associated to it. In this case, tuples are buffered and are only processed once the state of their key is received. This solution is preferable to suspending the stream as some stream sources do not support back pressure and would lose messages. To handle fault tolerance, the manager saves all routing configurations to stable storage before starting reconfiguration. If a POI crashes, the guarantees are the ones provided by the streaming engine and are not impacted by state migration.

```
manager_migration():
  for poi in POIS:
    send(poi, GET_METRICS, nil)                          ①

  for poi in POIS:
    metrics.add(receive(SEND_METRICS))                   ②

  reconf = compute_reconfiguration(metrics)

  for poi in POIS:
    send(poi, SEND_RECONF, reconf[poi])                  ③

  for poi in POIS:
    receive(ACK_RECONF)                                  ④

  for poi in POIS:
    if predecessors[poi].is_empty():
      send(poi, PROPAGATE, nil)                          ⑤

poi_migration():
  receive(GET_METRICS)                                   ①
  send(manager, SEND_METRICS, my_metrics)                ②

  my_reconf = receive(SEND_CONFIGURATION)                ③
  send(manager, ACK_RECONF, nil)                         ④

  if my_predecessors.is_empty():
    receive(PROPAGATE) # from manager
  else
    for poi in my_predecessors:
      receive(PROPAGATE)                                 ⑤

  update_routing(my_reconf[ROUTER])

  for (poi, keys) in my_reconf[SEND]:
    send(poi, MIGRATE, state(keys))                      ⑥

  for (poi, keys) in my_reconf[RECEIVE]:
    state.add_all(receive(MIGRATE))                      ⑥

  for poi in my_successors:
    send(poi, PROPAGATE, nil)                            ⑤
```

Algorithm 1: Pseudo-code for the reconfiguration algorithm

4. EVALUATION

In this section, we evaluate the benefits of locality-aware routing in stateful streaming applications by implementing it in Apache Storm. We first present the experimental setup. Then, we perform a thorough evaluation of application throughput when varying a wide range of configuration parameters using a synthetic workload. Afterwards, we switch to a real workload from Twitter to evaluate the benefits of an online approach in the case of a fluctuating workload. Finally, we evaluate the impact of reconfiguration on a real stable dataset from Flickr.

Summary of results.

We show using a synthetic workload that locality-aware routing significantly outperforms hash-based routing. The difference increases with the number of servers, the size of data tuples, and the amount of locality in the data. We observe on a Twitter dataset that associations between hashtags and locations vary over time. Our approach achieves a 50% locality on this dataset when deployed on 6 servers, compared to 17% for hash-based routing. We demonstrate that online reconfiguration is necessary to maintain this locality, as an offline approach is unable to leverage transient correlations. We also show that only the most frequent key associations are necessary to achieve high locality score, thereby confirming that 1MB of memory per POI is sufficient for collecting statistics. Finally, we observe the impact of online reconfiguration on a stable dataset from Flickr, and notice a significant throughput increase, thus validating our approach.

4.1 Experimental Setup

We implement locality-aware routing in Apache Storm in order to evaluate its performance. We now describe the testbed and the application deployed.

Servers.

Our platform for the experiments is a cluster of 9 physical servers. One of them runs Nimbus, i.e. the Apache Storm master, and the other 8 are Apache Storm workers. The workers are HPE Proliant DL380 Gen9 servers with the following specifications:

- **RAM** 128 GB of DDR4 memory.
- **CPU** 2x 10-Core Intel Xeon E5-2660v3@2.6 GHz
- **Network** The machines are linked through a 10 Gb/s network, using Jumbo frames (MTU 9000). We also experiment 1 Gb/s network speed by using a software to throttle bandwidth.
- **Disk** 15×6TB SATA disk.
- **Software** Debian 8.3.0 (stable), and Apache Storm 0.9.5.

Application.

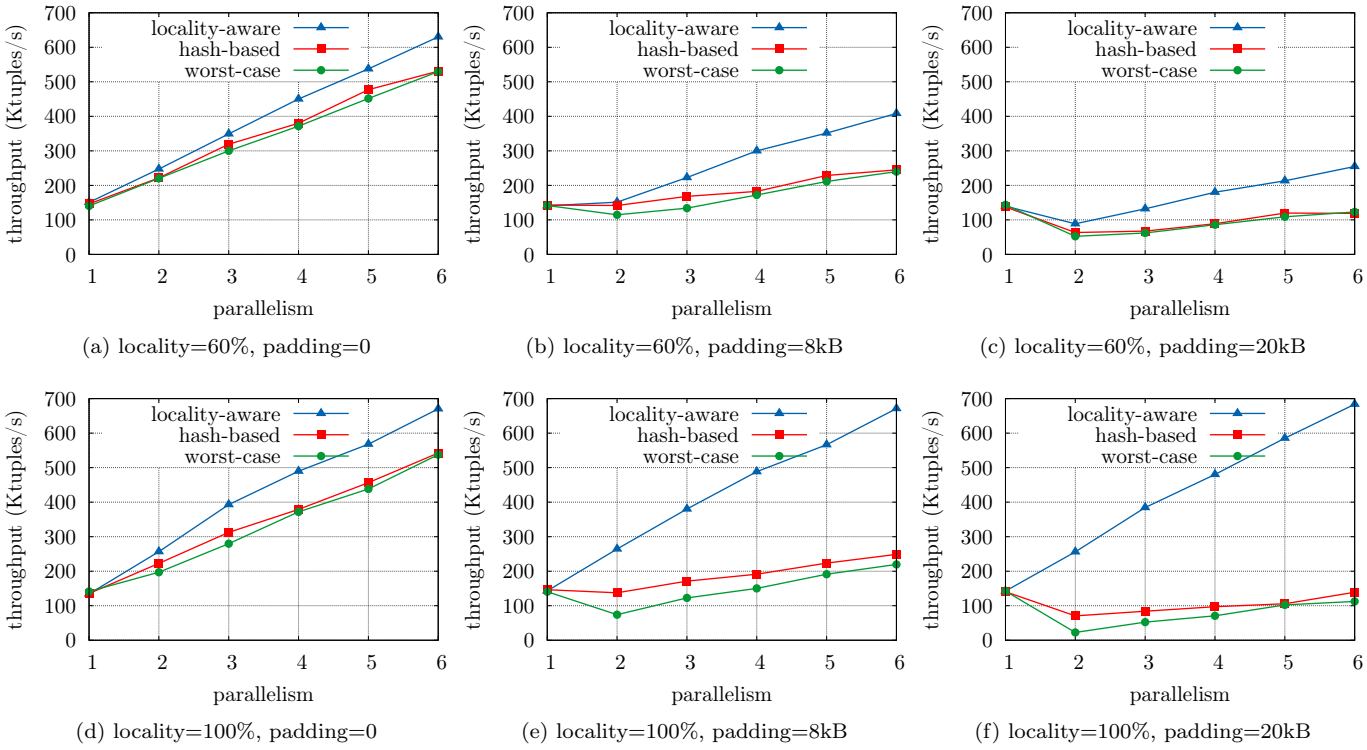
For evaluation purposes, we consider the case of a streaming application closely related to the one described in Section 3. It consists of one source, S, and two stateful POs, A and B. The first PO computes statistics based on the first field of the tuples by counting the number of occurrences of its different values, and the second PO executes the same operation on the second field. Hence, fields grouping is used to route data tuples to both POs. Each PO is deployed as n instances, n varying between 1 and 6. We later refer to the number of instances as the parallelism of an experiment. We use the same parallelism for both POs to ensure that every instance of the first PO has a local instance of the second. For each PO X, the POI X_i is hosted on server i. Hence, each server hosts both an instance of A and an instance of B.

4.2 Locality impact using synthetic workload

To assess the impact of locality on the performance of streaming applications, we first consider the case of a synthetic workload. The source generates tuples containing three fields: $(integer, integer, padding)$. The first two fields, the integers, vary between 1 and n. In this experiment, the *locality* parameter controls the number of tuples in which these integers are equal. The last field is here to allow experimenting with different tuple sizes, to simulate workloads ranging from single words to small texts. We vary the padding between 0 and 20kB.

We evaluate the performance of the application on this workload using 3 different versions of fields grouping.

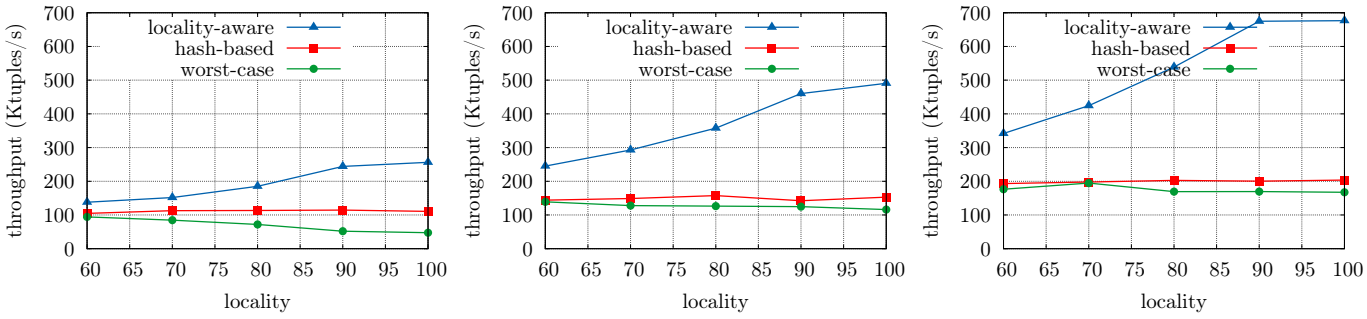
- *Hash-based:* tuples are assigned to a POI using a hash function on the key. This assignment is random but deterministic, and represents the default implementation of fields grouping in Storm.



(a) locality=60%, padding=0 (b) locality=60%, padding=8kB (c) locality=60%, padding=20kB

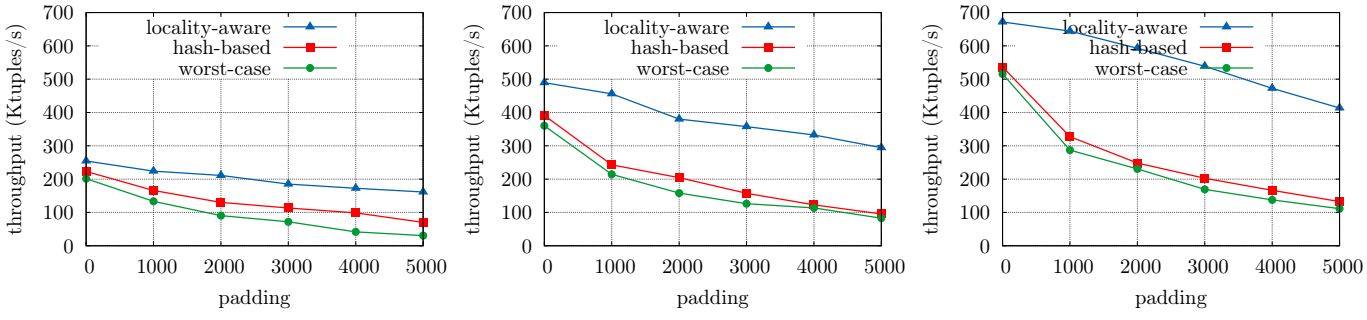
(d) locality=100%, padding=0 (e) locality=100%, padding=8kB (f) locality=100%, padding=20kB

Figure 7: Throughput when varying parallelism (number of machines), for different message sizes.



(a) parallelism=2, padding=12kB (b) parallelism=4, padding=12kB (c) parallelism=6, padding=12kB

Figure 8: Throughput when varying locality, with a message size of 12kB and different parallelisms



(a) locality=80%, parallelism=2 (b) locality=80%, parallelism=4 (c) locality=80%, parallelism=6

Figure 9: Throughput when varying tuple sizes, with a locality of 80% and different parallelisms

- *Locality-aware:* each tuple (i, j, p) is routed to the ith instance of A, A_i, and then to the jth instance of B, B_j. Doing so, all the tuples of type (i, i, p) are routed on the same server i. This implementation of fields grouping represents the approach advocated in the paper. These are the routing tables that would be generated by analysing the data.

- *Worst-case:* tuples of type (i, i, p) are always routed through the network. This represents a worst case scenario that has negative synergy with locality. This allows us to obtain a lower bound on the performance of the application.

We iterate over different values of following parameters: *parallelism*, *padding size*, and *locality*; and run the application with each of the 3 versions of fields grouping presented above.

Figure 7 depicts the throughput of the application when varying parallelism. With a locality of 60%, all versions of fields grouping send messages over the network. We observe that, as the size data tuples (padding) increases, the gains of adding additional servers decrease. When padding is at 20kB, we even notice a decrease of performance when switching from 1 to 2 servers. This behavior is symptomatic of stateful streaming applications: network constitutes a bottleneck, and the proportion of tuples transiting through the network increases with parallelism. For all padding sizes, *locality-aware* clearly outperforms the other options, as it is the only option that scales linearly for a parallelism higher than 2. A locality of 100% constitutes an ideal case, in which *locality-aware* avoids all network communications. In this setup, padding has no effect on the throughput as tuples are transferred in memory. This experiment highlights the impact of network communications on the throughput of streaming applications. Even when tuples are extremely small (padding = 0), routing through the network lowers the performance by 22%. As expected, this value increases with the size of tuples.

Figure 8 presents results for varying values of locality. *Hash-based* is not affected by locality, as it do not leverage it. *Locality-aware* gains performance as the locality of the dataset increases, since the amount of network communication decreases linearly with locality. We notice a plateau above 90% of locality.

Figure 9 illustrates the variation of throughput for varying padding sizes. The difference between *locality-aware* and the other options increases both with the size of padding, and with parallelism. This behavior is explained by the fact that in this experiment, *locality-aware* is able to preserve a ratio of 80% local communications, while *hash-based* sends more network messages as parallelism increases. Furthermore, network saturates faster when padding is high. We note that in the most challenging configurations, the performance of *hash-based* and *worst-case* are very similar.

4.3 Impact of online optimization

The approach presented in this paper is online. It can detect correlation between keys on a running application, and optimize locality even when the characteristic of data vary over time. In this section, we evaluate the benefits of online optimization over offline optimization. To this end, we rely on a real dataset from Twitter containing timestamped content.

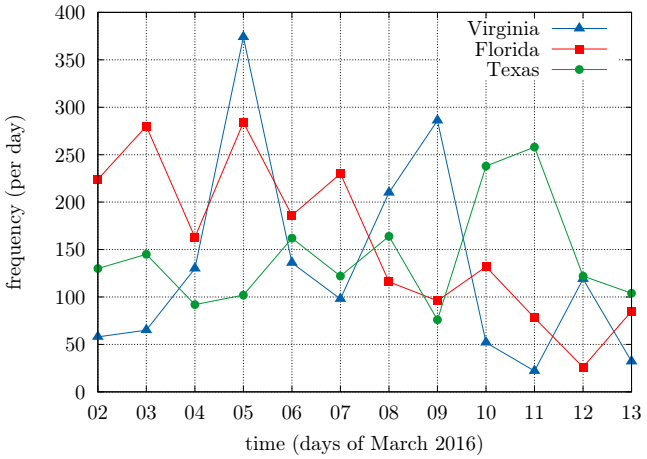


Figure 10: Occurrences of the hashtag *#nevertrump* in different states in the USA.

Twitter dataset.

We crawl tweets from October 2015 to May 2016 using the API of Twitter. Twitter provides for each tweet a location identifier which can be either the location of the user at the moment of the tweet, or a location associated to the content of the tweet. Locations can be countries, cities, or points of interests. Overall, our dataset contains 173 million associations between locations and hashtags. We set our application to first route using the location, and then the hashtag.

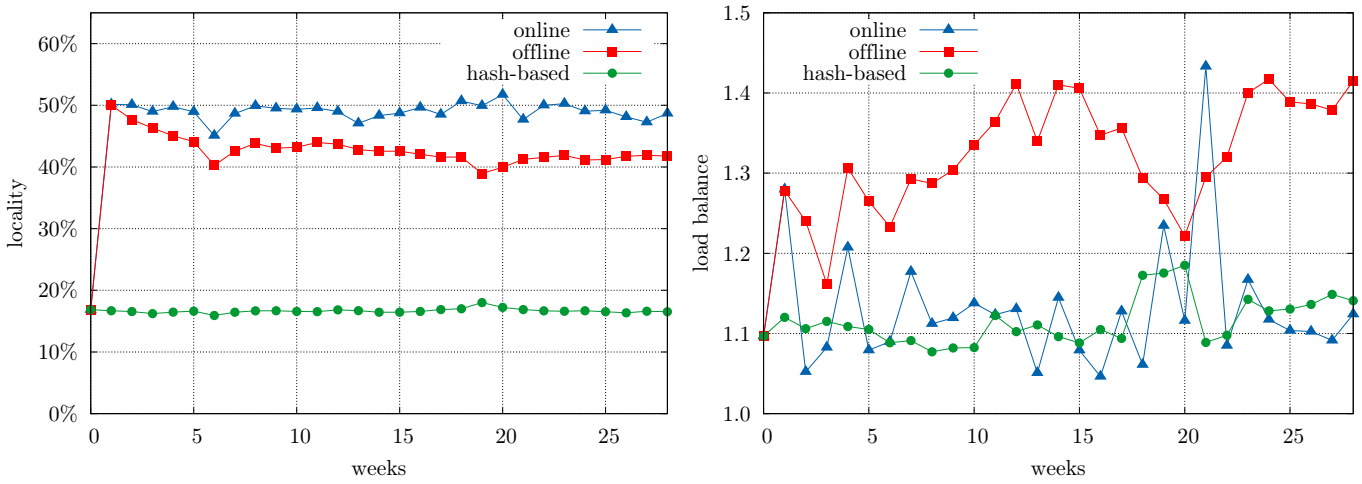
Evolution of correlations over time.

In social media, trends vary constantly and are often linked to events. Figure 10 shows the frequency of a popular hashtag on Twitter for different locations. While the tag appears on all of the three locations, it is clearly more correlated with Florida on March 3^{rd}, with Virginia on the 9^{th} and with Texas on the 11^{th}. In the context of our application, this means that, to optimize performance, the same hashtag should be co-located with 3 different locations at different periods of time. This justifies the online nature of our reconfiguration protocol: it is important to stay up-to-date regarding volatile correlations. The next experiment shows how reconfiguration affects the processing locality.

Online and offline optimization.

In this experiment, we compare the effectiveness of the *offline* approach, that computes a single configuration using a sample of data, and the *online* approach, that continuously updates the configuration. We use 1 week of data for the *offline* approach, while the *online* approach updates the configuration every week. We also present the performance of *hash-based* routing as a reference. We run this experiment with a parallelism of 6.

Figure 11a shows the evolution of locality over time, *Hash-based* achieves a locality of 16.6%, which corresponds to a random assignment with 6 servers. After one week, *online* and *offline* both obtain a sufficient amount of data to perform locality-aware routing, which raises the locality to 49%. However, this value decreases over time in the case of *offline*, and stabilizes around 40%. *Offline* preserves locality for stable associations, but fails to leverage transient ones. *Online*



(a) Locality (b) Load balance

Figure 11: Locality and load balance obtained after reconfiguration with a parallelism of 6, and period of one week. Online: reconfiguration every week. Offline: one reconfiguration after one week. Hash-based: no reconfiguration.

however maintains a locality in the vicinity of 50%. This shows that to capture volatile correlations, reconfiguration should be triggered on a regular basis. The experiments of Section 4.2 indicate that a 10% difference in locality can lead to a throughput gain of 25%. This demonstrates the benefits of the online optimization process in the case of fluctuating workloads. Note that when generating routing tables, Metis reports an expected locality of 75%. However, this locality is only achievable by running the exact dataset that was used to compute the configuration. In practice, data of the next week contains a significant proportion of new hashtags and locations that were not observed previously and are thus routed using hash functions.

Now that we have established the need for regular reconfiguration in order to optimize locality, we focus on the impact of reconfiguration on load balancing. As shown by Figure 11b, *hash-based* distributes the load fairly evenly, with an average of 12% additional traffic for the most loaded POI. *Online* and *offline* both start with a balanced load, thanks to the statistics collected on the frequency of keys. As the workload fluctuates, some hashtags and locations become more frequent in the following weeks. This causes the distribution of the load to deviate significantly. When optimizing for locality, correlated keys are assigned to the same server. While this contributes to diminishing network consumption and increasing throughput, this has a potential drawback: correlated keys have a higher probability to have peaks of activity simultaneously. *Online* is able to immediately correct these spikes of unbalance, while *offline* stabilizes around 40% imbalance. This experiment confirms that locality-aware routing is able to preserve load balance, and that reconfiguration needs to be carried out regularly on fluctuating workloads. We deliberately use long intervals of time (1 week) between reconfigurations for the online approach to highlight deviations from optimal behavior (imbalance spikes). In practice, as we show in Section 4.4, reconfiguration is extremely fast an can be triggered much more frequently to account for deviations in the frequency of keys.

Please note that the imbalance parameter α specified in Section 3.1 is indeed used and set to 1.03, which is Metis default value. However, while it is respected for the collected data on which the partitioning is achieved, its impact on the future data cannot be predicted, although the reconfiguration greatly improves the load balance as shown in Figure 11b.

Statistics collection.

As stated in Section 3.2, our online protocol uses a bounded amount of memory to collect statistics, and thus only retrieves information on the most frequent pairs of keys, or edges. This experiment aims at quantifying the impact of the number of considered top edges on the quality of the reconfiguration. Figure 12 shows how the achieved locality varies with the number of edges that we consider when performing the graph partitioning for different parallelisms. Naturally, having information on more pairs of keys results in better locality. However, considering the logarithmic scale of the figure, we can double the locality for parallelism 6, for instance, with only 0.1% of the total edges. Therefore, a quality/capacity trade-off is to be made when choosing the number of edges for the reconfiguration protocol, depending on the application and environment on which it will run. On this dataset, we find 10^6 edges is sufficient, so collecting statistics only occupies a few MB of memory of each POI.

4.4 Reconfiguration protocol validation

The following experiments launch the application with and without our reconfiguration protocol and see how the reconfiguration affects the application's performance. The tuples processed by our streaming application are of type $(tag, country, padding)$ which come from a dataset provided by Flickr and described below.

Flickr dataset.

This dataset [3] contains metadata about 100 million pictures posted on Flickr. Among other fields, there is a geolocation and a list of user tags for every picture. The geolocation is mapped to a country using data from Open-

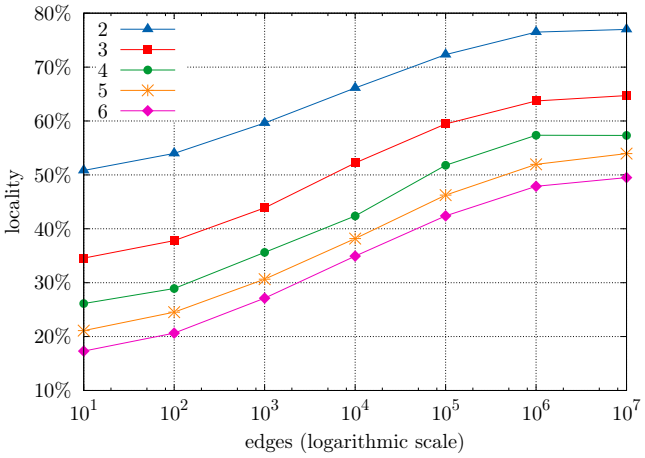


Figure 12: Locality achieved when varying number of considered edges, for different parallelisms.

StreetMap. This dataset represents a stable workload as there is no temporal information and images are not ordered.

Throughput analysis.

In this set of experiments, we launch our streaming application either without reconfiguration or with a reconfiguration every 10 minutes, each run lasts for 30 minutes. We do so for different tuple sizes (paddings) and using two different network settings, 10Gb/s and 1Gb/s. As shown by the different plots of Figure 13, a significant improvement of throughput follows the first reconfiguration at $t = 10min$ and is maintained throughout the run. This proves that the real-life correlation of Flickr data is sufficient to enhance performance through locality. As established in Section 4.2, the performance gain does indeed increase following the size of the tuples. By comparing results at 10Gb/s and results at 1Gb/s, we can see that the effect of tuple size is even greater in the case of a more limited bandwidth. As for the effect of parallelism, i.e., the number of instances of each PO, Figure 14 shows that the gap between throughput with and without reconfiguration is more important for higher numbers of instances. These executions on real workloads confirm the results obtained on synthetic workloads presented in Section 4.2. We also notice that deploying an updated configuration and migrating data is extremely fast and does not impact performance negatively, as the throughput increase is noticeable immediately after $t = 10min$.

5. RELATED WORK

5.1 Operator instance scheduling

When executing a streaming application, the scheduler deploys POIs on servers. The assignment of POIs to physical servers has a significant impact on the performance of the application. A first optimization criterion is ensuring that each server is assigned an even share of the computational load. A second objective is to locate tasks communicating frequently on the same servers to avoid network saturation.

In the stream processing engine `System S` [13], several operators are fused into a single processing element to achieve a good trade-off between communication cost and execution parallelism. The approach proposed is top-down, and starts by considering that all operators are part of the same processing element. This processing element is then recursively divided using graph partitioning algorithms.

Aniello et al. [5] proposed two schedulers for Storm. The offline scheduler only considers the topology of the application and tries to place consecutive POs on the same server. The online scheduler measures CPU and memory consumption of POIs, as well as communication patterns between POIs. POIs are then assigned to servers using a greedy algorithm, starting with the pair of POIs that communicate the most. The topology used for evaluation alternates shuffle grouping and fields grouping for routing, and fields grouping relies on hash functions only. The addition of local-or-shuffle grouping would significantly contribute to reducing network communications.

Fisher et al. [11] solve the scheduling problem using graph partitioning. POIs are vertices of the graph, and are weighted by the computational resources they consume. Edges represent communications between POIs, and are weighted by the amount of data that transits. In practice, the authors use Metis to obtain high quality partitions while preserving load balance.

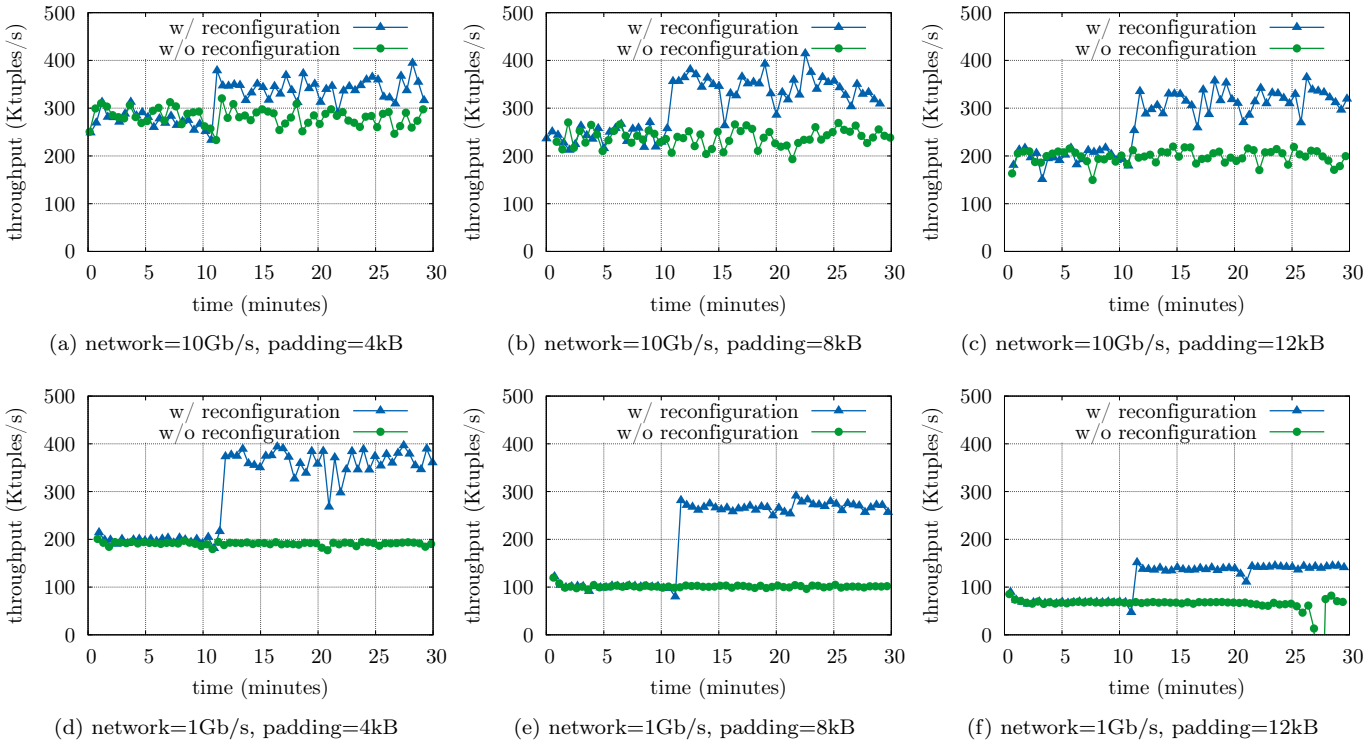
R-Storm [19] is a scheduler for Storm that aims at maximizing resource utilization while minimizing latency. The developer declares the memory and CPU utilization of each PO using a specific API. Then, a Knapsack-based heuristic performs the POI assignment. The authors describe two topologies typical of applications used by Yahoo in production. One of them is a chain, while the other starts with a transformation PO branching into two chains.

Cardellini et al. [8] proposed to perform an embedding of servers into a 4-dimensional cost-space. These dimensions represent network characteristics (latency and throughput), as well as processing power (utilization and availability). Coordinates are then used to optimize scheduling, using a spring-based formulation. The main novelty of this work is the use of the P2P algorithm Vivaldi to assess network latency in a distributed manner.

The problem of scheduling POIs is orthogonal to our contribution. We assume the existence of a scheduler that assigns POIs to servers, and take this assignment as an input constraint. While schedulers see fields grouping as a black box that cannot be optimized, our approach is able to improve data placement to further reduce network usage. Any online scheduler that actively measures communication between POIs can then notice the improvement and re-visit the POI placement decision, leading to even better performance. Our approach is similar to [11] as it relies on Metis for graph partitioning. Instead of considering a graph of POIs communicating, we consider a graph of keys that co-occur in the data.

5.2 Load balancing for stateful applications

Load balancing consists in ensuring that each server involved in a distributed system receives an even share of the total load to avoid bottlenecks. As explained in Section 2.2, stateless streaming applications rarely suffer from imbalanced load, as data tuples can be sent to any instance of a given PO. However stateful application use fields grouping to ensure that data tuples containing the same key always reach the same POI. In the case of skewed data distribution, POIs responsible for keys occurring frequently receive more tuples to process than other POIs, and become bottlenecks.



(a) network=10Gb/s, padding=4kB (b) network=10Gb/s, padding=8kB (c) network=10Gb/s, padding=12kB

(d) network=1Gb/s, padding=4kB (e) network=1Gb/s, padding=8kB (f) network=1Gb/s, padding=12kB

Figure 13: Evolution of the throughput with or without reconfiguration, for a parallelism of 6, different padding sizes and two types of network bandwidth.

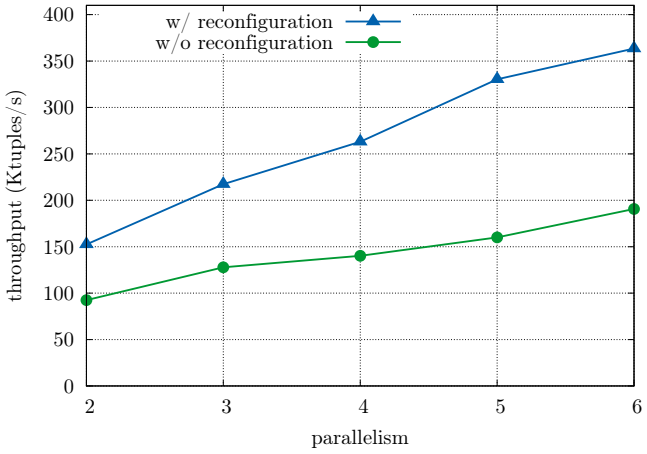


Figure 14: Average throughput for different parallelisms, and a padding of 4kB (on the 1Gb/s network). With reconfiguration, the average is measured after the first reconfiguration.

Several solutions have been proposed to limit the impact of data skew.

Nasir et al. [16] propose to use *partial* key grouping, where a key can be sent to two different POIs instead of one. Each POI locally estimates the load of its successors, and sends the data tuple to the least loaded of the two options. This solution is elegant, as it relies on two hash functions and does not require maintaining routing tables. However, it leads to additional memory consumption as the state associated to a given key is maintained by two POIs. This state is aggregated downstream, which limits the use of this solution to associative operators and introduces latency. Partial key grouping was improved to handle extremely frequent keys [17]. A list of the most frequent keys is maintained using the SpaceSaving algorithm [15]. These keys can be routed to any POI, instead of just two. This further increases memory consumption but improves load balancing in extreme cases.

Similarly to [17], Rivetti et al. [20] proposed DKG, an algorithm that maintains a list of the most frequent keys that cause load imbalance. These keys are then explicitly mapped to POIs using routing tables, while less frequent keys are routed using hash functions. This approach also relies on the SpaceSaving algorithm for estimating frequencies.

Skewed key distribution also cause load imbalance in database systems. E-store [21] keeps track of frequently accessed data tuples using SpaceSaving and migrates them in order to balance load between servers. A routing layer maintains the assignment of keys to servers using routing tables.

Our approach is similar to [17, 20, 21] as it relies on the SpaceSaving algorithm [15] to obtain an online estimation of frequency in data streams. However the statistics we collect are richer, as they involve pairs of keys. Hence, we are able to use them for network locality optimization in addition to load balancing. Moreover, our algorithm handles data migration in the case of a reconfiguration. This important problem is left to the application developers in [17, 20].

5.3 Co-locating correlated keys

Workload-driven optimization consists in analysing the workload of an application in order to tune the system to its

specific activity. Schism [9] analyses query logs of shared-nothing databases. Keys accessed by queries are represented as a graph, with edges weighted with the number of co-occurrences. This graph is partitioned using Metis to obtain a new assignment of keys to servers, such that each query can be answered with as few partitions as possible. Dynasore [6] performs similar optimizations for building social feeds. User profiles that are frequently accessed simultaneously are hosted on the same server. The offline partitioning relies on Metis, while an online optimizer reacts to workload changes dynamically. A streaming application can be interpreted as a continuous query, where operators (POIs) are placed on servers and data streams through them. We aim at ensuring that all POIs impacted by an execution query on a tuple of data are located on the same server. Hence, our approach is similar to [6, 9], as we analyse the workload to uncover correlations between keys, but is optimized for stream processing.

6. CONCLUSION

Stateful streaming applications suffer from increasing network consumption as they scale to multiple servers. To alleviate this drawback, we propose to increase data locality by explicitly routing correlated keys to the same servers. Our approach relies on lightweight statistics about co-occurrence of keys collected by stream operators. A manager gathers all statistics and performs a partitioning of the graph of keys to assign correlated keys to the same server while enforcing load balancing. Data related to reassigned keys is migrated following the order of operators in the application, which ensures that the state of a key is preserved. While in this work we only consider chains of POs, the same graph partitioning technique can be applied to more complex DAGs. When measuring association between keys, successor keys can be assigned to different POs, without changing the formulation of the problem. We demonstrate the effectiveness of our approach on synthetic and real datasets, by throughput increasing up to $\times 2$ with relatively small tuples. We prove the gain increases with the tuple size.

Our approach is able to deal with fluctuations in the correlations between keys by continuously optimizing routing. When the workload is very volatile, it is important to avoid triggering reconfigurations for ephemeral correlations, as the cost of reconfiguring would not be amortized. As future work, we will design estimators able to predict the impact of a reconfiguration to provide more fine-grained information to the manager. Another promising area of research is adapting this approach to hierarchical network structures. Instead of having a binary model in which keys are co-located or not, distances between servers can be taken into account to leverage rack locality when load balancing prevents server locality. This could be done by using hierarchical clustering, similarly to [6]. This however requires a larger testbed for validation.

7. ACKNOWLEDGMENTS

This work has been funded by the SMART SUPPORT CENTER (SSC) - RRA FEDER and by the FSN DATALYSE project.

We also thank HPE for lending us physical machines for our tests.

8. REFERENCES

[1] Apache Flink. https://flink.apache.org/.

[2] Apache Samza. http://samza.apache.org/.

[3] Flickr dataset. https://webscope.sandbox.yahoo.com/catalog.php?datatype=i&did=67.

[4] C. Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006.

[5] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM, 2013.

[6] X. Bai, A. Jégou, F. Junqueira, and V. Leroy. Dynasore: Efficient in-memory store for social applications. In *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*, pages 425–444, 2013.

[7] C. Budak, T. Georgiou, D. Agrawal, and A. El Abbadi. Geoscope: Online detection of geo-correlated information trends in social networks. *Proc. VLDB Endow.*, 7(4):229–240, Dec. 2013.

[8] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 344–347, New York, NY, USA, 2015. ACM.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[11] L. Fischer and A. Bernstein. Workload scheduling in distributed stream processors using graph partitioning. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 124–133, 2015.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[13] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 16:1–16:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[14] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.

[15] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both

choices: Practical load balancing for distributed stream processing engines. In *31st IEEE International Conference on Data Engineering, ICDE*, pages 137–148, 2015.

[17] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *32nt IEEE International Conference on Data Engineering, ICDE*, 2015.

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[19] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, New York, NY, USA, 2015. ACM.

[20] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 80–91, New York, NY, USA, 2015. ACM.

[21] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proc. VLDB Endow.*, 8:245–256, November 2014.

[22] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[23] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[24] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.