Graham Cormode Ke Yi
Antonios Deligiannakis Minos Garofalakis(Eds.)

First International Workshop on Big Dynamic Distributed Data (BD3)

Workshop at VLDB 2013
Riva Del Garda, Italy, August 30, 2013

Proceedings

Editors' contacts:
G.Cormode@warwick.ac.uk, yike@cse.ust.hk, adeli@softnet.tuc.gr, minos@softnet.tuc.gr

Preface

As the amount of streaming data produced by large-scale systems such as environmental monitoring, scientific experiments and communication networks grows rapidly, new approaches are needed to effectively process and analyze such data. There are several promising directions in the area of large-scale distributed computation, that is, where multiple computing entities work together over partitions of the massive, streaming data to perform complex computations. Two important paradigms in this realm are continuous distributed monitoring (i.e., continually maintaining an accurate estimate of a complex query), and distributed and cluster-based systems that allow the processing of big, streaming data (e.g., IBM System S, Apache S4, and Twitter Storm).

The aim of the BD3 workshop is to bring together computer scientists with interests in this field to present recent innovations, find topics of common interest and to stimulate further development of new approaches to deal with massive dynamic and distributed data.

August 2013

Graham Cormode, Antonios Deligiannakis,
Minos Garofalakis, Ke Yi

Organizing Committee

General Chairs:
Minos Garofalakis
Technical University of Crete
minos@softnet.tuc.gr

Antonios Deligiannakis
Technical University of Crete
adeli@softnet.tuc.gr

Program Chairs:
Graham Cormode
University of Warwick
G.Cormode@warwick.ac.uk

Ke Yi
Hong Kong University of Science and Technology
yike@cse.ust.hk

Publicity Chair:
Odysseas Papapetrou
Technical University of Crete
papapetrou@softnet.tuc.gr

Program Committee

Alin Dobra	U. Florida
Pascal Felber	Universite de Neuchatel
Christof Fetzer	TU Dresden
Ling Huang	Intel Research
Daniel Keren	Haifa
Andrew McGregor	UMass-Amherst
Stavros Papadopoulos	HKUST
Odysseas Papapetrou	Technical University of Crete
Jeff Phillips	Utah
Peter Pietzuch	Imperial College London
Neoklis Polyzotis	UC Santa Cruz
Assaf Schuster	Technion
Izchak Sharfman	Technion
Nesime Tatbul	Intel Labs / MIT
Srikanta Tirthapura	Iowa State
Suresh Venkatasubramanian	Utah
Milan Vojnovic	Microsoft Research
Qin Zhang	IBM Research

Contents

Safe-Zones for Monitoring Distributed Streams

Daniel Keren
Haifa University

Guy Sagy
Technion

Amir Abboud
Technion

David Ben-David
Technion

Izchak Sharfman
Technion

Assaf Schuster
Technion

ABSTRACT

In many emerging applications, the data which has to be monitored is of very high volume, dynamic, and distributed, making it infeasible to collect the distinct data streams to a central node and process them there. Often, the monitoring problem consists of determining whether the value of a global function, which depends on the union of all streams, crossed a certain threshold. A great deal of effort is directed at reducing communication overhead by transforming the monitoring of the global function to the testing of *local* constraints, checked independently at the nodes. Recently, *geometric monitoring* (GM) proved to be very useful for constructing such local constraints for general (non-linear, non-monotonic) functions. Alas, in all current variants of geometric monitoring, the constraints at all nodes share an identical structure and are, thus, unsuitable for handling heterogeneous streams, which obey different distributions at the distinct nodes. To remedy this, we propose a general approach for geometric monitoring of heterogeneous streams (HGM), which defines constraints tailored to fit the distinct data distributions at the nodes. While optimally selecting the constraints is an NP-hard problem, we provide a practical solution, which seeks to reduce running time by hierarchically clustering nodes with similar data distributions and then solving more, but simpler, optimization problems. Experiments are provided to support the validity of the proposed approach.

1. INTRODUCTION

For a few years now, processing and monitoring of distributed streams has been emerging as a major effort in data management, with dedicated systems being developed for the task [1]. This paper deals with *threshold queries* over distributed streams, which are defined as "retrieve all items x for which $f(x) \leq T$", where $f()$ is a scoring function and T some threshold. Such queries are the building block for many algorithms, such as top-k queries, anomaly detection, and system monitoring. They are also applied in important data processing and data mining tools, including feature selection, decision tree construction, association rule mining, and computing correlations. Another important application is data classification, which is often also achieved by thresholding a function, such as the output of a neural net or support vector machine.

Geometric monitoring (GM) [2, 3, 4, 5] has been recently proposed for handling such threshold queries over distributed data. While a more detailed presentation is deferred until Section 2, we note that GM can be applied to the important case of scoring functions $f()$ evaluated at the average of dynamic data vectors $v_1(t), \ldots, v_n(t)$, maintained at n distributed nodes. Here, $v_i(t)$ is an m-dimensional data vector, often denoted as *local vector*, at the i-th node N_i at time t (often t will be suppressed). In a nutshell, each node monitors a convex subset, often referred to as the node's *safe-zone*, of the *domain* of these data vectors, as opposed to their *range*. What is guaranteed in GM is that the global function $f()$ will not cross its specified threshold as long as all data vectors lie within their corresponding safe-zones. Thus, each node remains silent as long as its data vector lies within its safe zone. Otherwise, in case of a safe-zone breach, communication needs to take place in order to check if the function has truly crossed the given threshold.

The geometric technique can support any scoring function $f()$, evaluated at the average of the dynamic data vectors. Thus, $f()$ is not assumed to obey some simple property (e.g., linearity or monotonicity). To add to the generality of the technique [3, 6], the v_i vector can contain not only the raw data, but any function (i.e., norm, logarithm, power, variance, etc) computed over the data of N_i. Thus, GM allows the monitoring of functions that are far more complex and general than simple aggregates.

A crucial component for reducing the communication required by the geometric method is the design of the safe-zone in each node. Nodes remain silent as long as their local vectors remain within their safe-zone. Thus, good safe-zones increase the probability that nodes will remain silent, while also guaranteeing correctness: a global threshold violation cannot occur unless at least one node's local vector lies outside the corresponding node's safe-zone.

However, prior work on geometric monitoring has failed to take into account the nature of heterogeneous data streams, in which the data distribution of the local vectors at different nodes may vary significantly. This has led to a uniform treatment of all nodes, independently of their characteristics, and the assignment of identical safe-zones (i.e., of the same shape and size) to all nodes.

As we demonstrate in this paper, designing safe-zones that take into account the data distribution of nodes can lead to efficiently monitoring threshold queries at a fraction (requiring an order of magnitude fewer messages) of what prior techniques achieve. However, designing different safe-zones for the nodes is by no means an easy task. In fact, the problem is NP-hard (proof omitted due to lack of space). We thus propose a more practical solution that hierarchically clusters nodes, based on the similarity of their data distributions, and then seeks to solve many small (and easier) optimization problems.

The contributions of this work are:

- We formulate a far more general safe-zone assignment problem than those which were treated so far. Instead of constructing one safe-zone which is common to all nodes, we seek to fit each node with a safe-zone that suits its data distribution.
- We present a practical solution, which uses hierarchical clustering of the nodes to construct the safe-zones, while applying various geometric and computational tools.
- The resulting safe-zones were tested on real data, where we demonstrate that: (i) the hierarchical clustering approach dramatically reduces the running time of the safe-zone assignment process, (ii) our techniques may result in one order of magnitude (or even larger) improvements in communication over previous geometric monitoring methods, even for a small number of nodes.

Outline. We survey prior work on the geometric approach in Section 2. In Section 3 we formulate our optimization problem, which involves the design of safe-zones at the nodes. Section 4 presents our algorithmic framework. Experiments are resented in Section 5. Lastly, conclusions are offered.

Hereafter we denote our proposed method for geometric monitoring of heterogeneous streams as **HGM**, in contrast to prior work on geometric monitoring that is denoted **GM**.

2. RELATED WORK

Space limitations allow us to only survey previous work on geometric monitoring (GM). We now describe some basic ideas and concepts of the GM technique, which was introduced and applied to monitor distributed data streams in [2, 7].

As described in Section 1, each node N_i maintains a local vector v_i, while the monitoring function $f()$ is evaluated at the average v of the v_i vectors. Before the monitoring process, each node N_i is assigned a subset of the data space, denoted as S_i – its *safe-zone* – such that, as long as the local vectors are inside their respective safe-zones, it is guaranteed that the global function's value did not cross the threshold; thus the node remains silent as long as its local vector v_i is inside S_i. If $v_i \notin S_i$ (local violation), a violation recovery ("balancing") algorithm [2] can be applied.

For details and scope of GM see [5] and the survey in [8]. Recently, GM was successfully applied to detecting outliers in sensor networks [3], extended to prediction-based monitoring [4], and applied to other monitoring problems [9, 10, 11].

Basic definitions relating to GM. A basic construct is the *admissible region*, defined by $A \triangleq \{v|f(v) \leq T\}$. Since the value we wish to monitor is $f\left(\frac{v_1+...+v_n}{n}\right)$, any viable assignment of safe-zones must satisfy $\bigwedge_{i=1}^{n}(v_i \in S_i) \rightarrow v = (v_1 + ... + v_n)/n \in A$. This guarantees that as long as all nodes are silent, the average of the v_i vectors remains in A and, therefore, the function has not crossed the threshold. The question is, of course, how to determine the safe-zone S_i of each node N_i; in a sense to be made precise in Section 3, it is desirable for the safe-zones to be as large as possible.

In [5] it was proved that all existing variants of GM share the following property: each of them defines some convex subset C of A (different methods induce different C's), such that each safe-zone S_i is a translation of C – that is, there exist vectors u_i $(1 \leq i \leq n)$ such that $S_i = \{u_i + c|c \in C\}$ and $\sum_{i=1}^{n} u_i = 0$. This observation unifies the distinct variants of GM, and also allows to easily see why $\bigwedge_{i=1}^{n}(v_i \in S_i)$ implies that $v \in C \subseteq A$ – it follows immediately from the fact that convex subsets are closed under taking averages and from the fact that the u_i vectors sum to zero.

Here we assume that C is given; it can be provided by any of the abovementioned methods (obviously, if A itself is convex, we just choose $C = A$). We propose to extend previous work in a more general direction. Our goal here is to handle a basic problem which haunts all the existing GM variants: *the shapes of the safe-zones at different nodes are identical.* Thus, if the data is heterogeneous across the distinct streams (an example is depicted in Figure 1), meaning that the data at different nodes obeys different distributions, existing GM algorithms will perform poorly, causing many local violations that do not correspond to global threshold crossing ("false alarms").

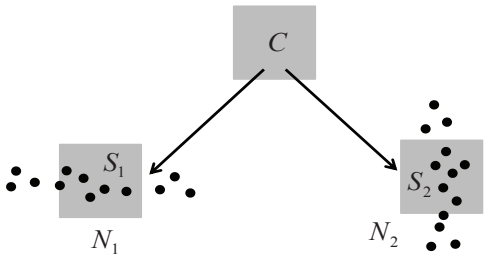


Figure 1: Why GM may fail for heterogeneous streams. Here C is equal to a square, and the data distribution at the two nodes is schematically represented by samples. In GM, the safe-zones at both nodes are restricted to be a translation of C, and thus cannot cover the data; HGM will allow much better safe-zones (see Section 3, and Figure 2).

In this paper, we present a more general approach that allows to assign *differently shaped* safe-zones to different nodes. Our approach requires tackling a difficult optimization problem, for which practical solutions need to be devised.

3. SAFE-ZONE DESIGN AS AN OPTIMIZATION PROBLEM

We now seek to formulate an optimization problem, whose solution defines the safe-zones at all nodes. The safe-zones should satisfy the following properties:

Correctness: If S_i denotes the safe-zone at node N_i, we must have: $\bigwedge_{i=1}^{n} (v_i \in S_i) \rightarrow (v_1 + ... + v_n)/n \in C$. This ensures that every threshold crossing by $f(v)$ will result in a safe-zone breach in at least one node.

Expansiveness: Every safe-zone breach (local violation) triggers communication, so the safe-zones should be as "large" as possible. We measure the "size" of a safe-zone S_i by its probability volume, defined as $\int_{S_i} p_i(v)dv$ where p_i is the pdf of the data at node N_i. Probabilistic models have proved useful in predicting missing and future stream values in various monitoring and processing tasks [12, 13, 14], including previous geometric methods [5], and their incorporation in our algorithms proved useful in monitoring real data (Section 5). To handle these two requirements, we formulate a constrained optimization problem as follows:

Given: (1) probability distribution functions $p_1, \ldots, p_n$ at n nodes
(2) A convex subset C of the admissible region A

$$\text{Maximize} \int_{S_1} p_1 dv_1 \cdot ... \cdot \int_{S_n} p_n dv_n \quad \text{(expansiveness)}$$

$$\text{Subject to } \frac{S_1 \oplus \cdots \oplus S_n}{n} \subseteq C \quad \text{(correctness)}$$

where $\frac{S_1 \oplus \cdots \oplus S_n}{n} = \left\{ \frac{v_1 + \cdots + v_n}{n} \,|v_1 \in S_1, \ldots, v_n \in S_n \right\}$, or the *Minkowski sum* [15] of $S_1, \ldots, S_n$, in which every element is divided by n (the *Minkowski average*). Introducing the Minkowski average is necessary in order to guarantee correctness, since v_i must be able to range over the entire safe-zone S_i. Note that instead of using the constraint $\frac{S_1 \oplus ... \oplus S_n}{n} \subseteq A$, we use $\frac{S_1 \oplus ... \oplus S_n}{n} \subseteq C$. This preserves correctness, since $C \subseteq A$. The reason we chose to use C is that typically it's much easier to check the constraint for the Minkowski average containment in a convex set; this is discussed in Section 4.4.

To derive the target function $\int_{S_1} p_1 dv_1 \cdot ... \cdot \int_{S_n} p_n dv_n$, which estimates the probability that the local vectors of all nodes will remain in their safe-zones, we assumed that the data is not correlated between nodes (hence we multiply the individual probabilities), as it was the case in the experiments in Section 5 (see also [13] and the discussion therein). If the data is correlated, the algorithm is essentially the same, with the expression for the probability that data at some node breaches its safe-zone modified accordingly.

Note that correctness and expansiveness have to reach a "compromise": figuratively speaking, the correctness constraint restricts the size of the safe-zones, while the probability volume increases as the safe-zones become larger. This trade-off is central in the solution of the optimization problem.

The advantage of the resulting safe-zones is demonstrated by a schematic example (Figure 2), in which C and the stream pdfs are identical to those in Figure 1. In HGM, however, the individual safe-zones can be shaped very differently from C, allowing a much better coverage of the pdfs, while adhering to the correctness constraint. Intuitively speaking, nodes can trade "geometric slack" between them; here S_1 trades "vertical slack" for "horizontal slack".

4. CONSTRUCTING THE SAFE-ZONES

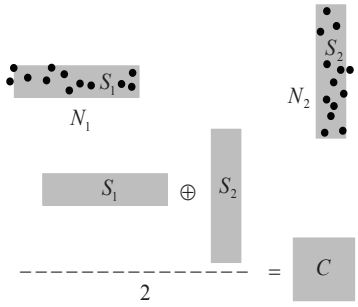


Figure 2: Schematic example of HGM safe-zone assignment for two nodes, which also demonstrates the advantage over previous work. The convex set C is a square, and the pdf at the left (right) node is uniform over a rectangle elongated along the horizontal (vertical) direction. HGM can handle this case by assigning the two rectangles S_1, S_2 as safe-zones, which satisfies the correctness requirement (since their Minkowski average is equal to C). GM (Figure 1) will perform poorly in this case.

We now briefly describe the overall operation of the distributed nodes. The computation of the safe-zones is initially performed by a coordinator node, using a process described in this section. This process is performed infrequently, since there is no need to change the safe-zones of a node unless a global threshold violation occurs. As described in Section 3, the input to the algorithm is: (1) The probability distribution functions $p_1, \ldots, p_n$ at the n nodes. These pdfs can be of any kind (e.g., Gaussian [7], random walk [16], uniform, etc). (2) A convex subset C of the admissible region A.

Given this input, the coordinator applies the algorithm described in Sections 4.1 to 4.5 to compute $S_1 ... S_n$ which solve the optimization problem defined in Section 3. Then, node N_k is assigned S_k.

4.1 Solving the Optimization Problem

In order to efficiently solve our optimization problem, we need to answer several questions:

- What kinds of shapes to consider for candidate safe-zones? This is discussed in Section 4.2.
- The target function is defined as the product of integrals of the respective pdfs on the candidate safe-zones. Given candidate safe-zones, how do we efficiently compute the target function? This is discussed in Section 4.3.
- Given candidate safe-zones, how do we efficiently test if their Minkowski average lies in C? This is discussed in Section 4.4.
- As we will point out, the number of variables to optimize over is very large, with this number increasing with the number of nodes. It is well-known that the computational cost of general optimization routines increases at a super-linear rate with the number of variables. To remedy this issue, we propose in Section 4.5 a hierarchical clustering approach, which uses a divide-and-conquer algorithm to reduce the problem to that of recursively computing safe-zones for small numbers of nodes.

4.2 Shape of Safe-Zones to Consider

The first step in solving an optimization problem is determining the parameters to optimize over. Here, the space of parameters is huge – *all* subsets of the Euclidean space are candidates for safe-zones. For one-dimensional (scalar) data, intervals provide a reasonable choice for safe-zones, but for higher dimensions no clear candidate exists.

To achieve a practical solution, we choose the safe-zones from a parametric family of shapes, denoted by S. This family of shapes should satisfy the following requirements:

- It should be broad enough so that its members can reasonably approximate every subset which is a viable candidate for a safe-zone.
- The members of S should have a relatively simple shape. In practice, this means that they are polytopes with a restricted number of vertices, or can be defined by a small number of implicit equations (e.g., polynomials [17]).
- It should not be too difficult to compute the integral of the various pdfs over members of S (Section 4.3).
- It should not be too difficult to compute, or bound, the Minkowski average of members of S (Section 4.4).

The last two conditions allow efficient optimization. If computing the integrals of the pdf or the Minkowski average are time consuming, the optimization process may be lengthy. We thus considered and applied in our algorithms various polytopes (such as triangles, boxes, or more general polytope) as safe-zones; this yielded good results in [5].

The choices of S applied here have provided good results in terms of safe-zone simplicity and effectiveness. However, the challenge of choosing the best shape for arbitrary functions and data distributions is quite formidable, and we plan to continue studying it in the future.

4.3 Computing the Target Function

The target function is defined as the product of integrals of the respective pdfs on the candidate safe-zones. Typically, data is provided as discrete samples. The integral can be computed by first approximating the discrete samples by a continuous pdf, and then integrating it over the safe-zone. We used this approach, fitting a GMM (Gaussian Mixture Model) to the discrete data and integrating it over the safe-zones, which were defined as polytopes. To accelerate the computation of the integral, we used Green's Theorem to reduce a double integral to a one-dimensional integral over the polygon's boundary, for the two-dimensional data sets in the experiments. For higher dimensions, the integral can also be reduced to integrals of lower dimensions, or computed using Monte-Carlo methods.

4.4 Checking the Constraints

A simple method to test the Minkowski sum constraint relies on the following result [18]:

LEMMA 1. *If P and Q are convex polytopes with vertices $\{P_i\}$, $\{Q_j\}$, then $P \oplus Q$ is equal to the convex hull of the set $\{P_i + Q_j\}$.*

Now, assume we wish to test whether the Minkowski average of P and Q is contained in C. Since C is convex, it contains the convex hull of every of its subsets; hence it suffices to test whether the points $(P_i + Q_j)/2$ are in C, for all i, j. If not all points are inside C, then the constraint violation can be measured by the maximal distance of a point $(P_i + Q_j)/2$ from C's boundary. The method easily generalizes to more polytopes: for three polytopes it is required to test the average of all triplets of vertices, etc.

4.5 Hierarchical Clustering

While the algorithms presented in Sections 4.3-4.4 reduce the running time for computing the safe-zones, our optimization problem still poses a formidable difficulty. For example, fitting octagonal safe-zones [5] to 100 nodes with two-dimensional data requires to optimize over 1,600 variables (800 vertices in total, each having two coordinates), which is quite high. To alleviate this problem, we first organize the nodes in a hierarchical structure, which allows us to then solve the problem recursively (top-down) by reducing it to sub-problems, each containing a much smaller number of nodes.

We first perform a bottom-up hierarchical clustering of the nodes. To achieve this, a distance measure between nodes needs to be defined. Since a node is represented by its data vectors, a distance measure should be defined between subsets of the Euclidean space. We apply the method in [19], which defines the distance between sets by the L^2 distance between their moment vectors (vectors whose coordinates are low-order moments of the set). The moments have to be computed only once, in the initialization stage. The leaves of the cluster tree are individual nodes, and the inner vertices can be thought of as "super nodes", each containing the union (Minkowski average) of the data of nodes in the respective sub-tree. Since the moments of a union of sets are simply the sum of the individual sets' moments, the computation of the moment for the inner nodes is very fast.

After the hierarchical clustering is completed, the safe-zones are assigned top-down: first, the children of the root are assigned safe-zones under the constraint that their Minkowski average is contained in C. In the next level, the grandchildren of the root are assigned safe-zones under the constraint that their Minkowski average is contained in their parent nodes' safe-zones, etc. The leaves are either individual nodes, or clusters which are uniform enough and can all be assigned safe-zones with identical shapes.

5. EXPERIMENTS

HGM was implemented and compared with the GM method, as described in [5], which is the most recent variant of previous work on geometric monitoring that we know of. We are not aware of other algorithms which can be applied to monitor the functions treated here (the ratio queries in [20] deal with accumulative ratios and not instantaneous ones as in our experiments).

5.1 Data, Setup and Monitored Functions

5.1.1 Data and Monitored Functions

Our data consists of air pollutant measurements taken from "AirBase – The European Air Quality Database" [21], measured in micrograms per cubic meter. Nodes correspond to sensors at different geographical locations. The data at different nodes greatly varies in size and shape and is irregular as a function of time. The monitored functions were chosen due to their practical importance, and also as they are non-linear and non-monotonic and, thus, cannot be handled by most existing methods. In Section 5.2 results are presented for monitoring the ratio of NO to NO_2,

which is known to be an important indicator in air quality analysis [22]. An example of monitoring a quadratic function in three variables is also presented (Section 5.3); quadratic functions are important in numerous applications (e.g., the variance is a quadratic function in the variables, and a normal distribution is the exponent of a quadratic function, hence thresholding it is equivalent to thresholding the quadratic).

5.1.2 Choosing the Family of Safe-Zones

To solve the optimization problem, it is necessary to define a parametric family of shapes S from which the safe-zones will be chosen. Section 4.2 discusses the properties this family should satisfy. In [5], the suitability of some families of polytopes is studied for the simpler, but related, problem of finding a safe-zone common to all nodes. The motivations for choosing S here were:

- Ratio queries (Section 5.2) – the triangular safe-zones (Figure 3) have the same structure, but not size or location, as C, and are very simple to define and apply.

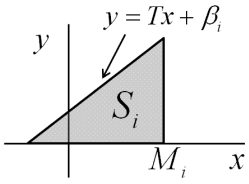


Figure 3: Triangular safe-zones used for ratio monitoring.

- Quadratic function (Section 5.3) – here we allowed general polytopes, and tested the results for increasing numbers of vertices. The model selected was with 12 vertices, in which the target function to optimize was "saturated" (i.e. adding more vertices increased the value by less than 0.1%).

5.1.3 Optimization Parameters and Tools

The triangular safe-zones (Section 5.2) have two degrees of freedom each (M_i and β_i, see Figure 3), hence for n nodes we have $2n$ parameters to optimize over. The safe-zones in Section 5.3 require 36 parameters each. In all cases we used the Matlab routine fmincon to solve the optimization problem [23]. To compute the integral of the pdf on the safe-zones, data was approximated by a Gaussian Mixture Model (GMM), using a Matlab routine [24].

5.2 Ratio Queries

This set of experiments concerned monitoring the ratio between two pollutants, NO and NO_2, measured in distinct sensors. Each of the n nodes holds a vector (x_i, y_i) (the two concentrations), and the monitored function is $\frac{\sum y_i}{\sum x_i}$ (in [20] ratio is monitored but over aggregates over time, while here we monitor the instantaneous ratio for the current readings). An alert must be sent whenever this function is above a threshold T (taken as 4 in the experiments), and/or when the NO_2 concentration is above 250. The admissible region A is a triangle, therefore convex, so $C = A$. The safe-zones tested were triangles of the form depicted in Figure 3, a choice motivated by the shape of C. The half-planes method (Section 4.4) was used to test the constraints. An example with four nodes, which demonstrates the advantage of allowing different safe-zones at the distinct nodes, is depicted in Figure 4.

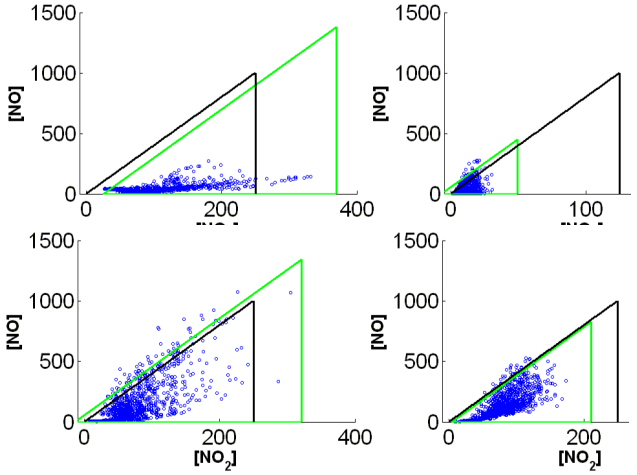


Figure 4: Example of safe-zones with four nodes. The convex set C is the triangle outlined in black, safe-zones are outlined in green. Nodes with more compact distributions are assigned smaller safe-zones, and nodes with high values of the monitored function (NO/NO_2 ratio) are assigned safe-zones which are translated to the left in order to cover more data. This is especially evident in the top right node, in which the safe-zone is shifted to the left so it can cover almost all the data points. In order to satisfy the Minkowski sum constraint, the safe-zone of top left node is shifted to the right, which in that node hardly sacrifices any data points; also, the larger safe-zones are balanced by the smaller ones. Note that HGM allows safe-zones which are *larger* than the admissible region A, as opposed to previous work, in which the safe-zones are subsets of A.

Improvement Over Previous GM Work. We compared HGM with GM in terms of the number of produced local violations. In Figure 5, the number of safe-zone violations is compared for various numbers of nodes. HGM results in significantly fewer local violations, even for a small number of nodes. As the number of nodes increases, the benefits of HGM over GM increase. For a modest network size of 10 nodes, HGM requires less than an order of magnitude fewer messages than GM.

5.3 Monitoring a Quadratic Function

Another example consists of monitoring a quadratic function with more general polyhedral safe-zones in three variables (Figure 6). The data consists of measurements of three pollutants (NO, NO_2, SO_2), and the safe-zones are polyhedra with 12 vertices. The admissible region A is the ellipsoid depicted in pink; since it is convex, $C = A$. As the extent of the data is far larger than A, the safe-zones surround the regions in which the data is denser. Here we did not compare to previous methods.

6. CONCLUSIONS AND FUTURE WORK

An approach for minimizing communication while monitoring threshold queries over heterogeneous distributed streams

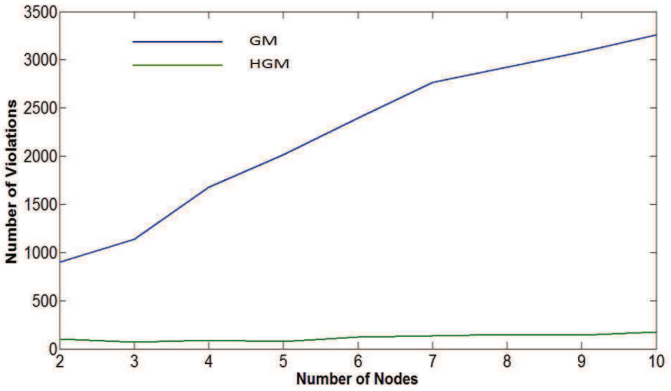


Figure 5: Comparison of our HGM (green) to GM [5] (blue) in terms of number of violations, up to 10 nodes.

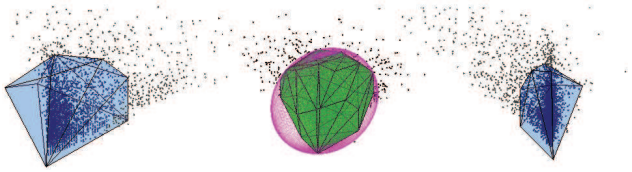


Figure 6: Monitoring a quadratic function. The set C is the pink ellipsoid, the safe-zones are polyhedra with 12 vertices each (in pale blue), and their Minkowski average is in green.

was presented. It is formulated as an optimization problem of a geometric and probabilistic flavor, whose solution assigns each node a "safe-zone" with the property that a node may remain silent as long as its data vector is in its safe-zone. While the problem is known to be difficult, a practical solution using a hierarchical clustering algorithm is presented and implemented for two and three dimensional data, allowing to achieve substantial improvement over previous work, while using rather simple safe-zones which also reduce the computational effort at the nodes.

7. ACKNOWLEDGMENT

This work was partially supported by the European Commission under ICT-FP7-LIFT-255951 (Local Inference in Massively Distributed Systems).

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *CIDR*, 2005.

[2] I. Sharfman, A. Schuster, and D. Keren, "A geometric approach to monitoring threshold functions over distributed data streams," *ACM Trans. Database Syst.*, vol. 32, no. 4, 2007.

[3] S. Burdakis and A. Deligiannakis, "Detecting outliers in sensor networks using the geometric approach," in *ICDE*, 2012.

[4] N. Giatrakos, A. Deligiannakis, M. N. Garofalakis, I. Sharfman, and A. Schuster, "Prediction-based geometric monitoring over distributed data streams," in *SIGMOD*, 2012.

[5] D. Keren, I. Sharfman, A. Schuster, and A. Livne, "Shape sensitive geometric monitoring," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 8, 2012.

[6] G. Sagy, D. Keren, I. Sharfman, and A. Schuster, "Distributed threshold querying of general functions by a difference of monotonic representation," *PVLDB*, vol. 4, no. 2, 2010.

[7] I. Sharfman, A. Schuster, and D. Keren, "Shape sensitive geometric monitoring," in *PODS*, 2008.

[8] G. Cormode, "Algorithms for continuous distributing monitoring: A survey," in *AlMoDEP*, 2011.

[9] J. Kogan, "Feature selection over distributed data streams through optimization," in *SDM*, 2012.

[10] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis, "Sketch-based querying of distributed sliding-window data streams," *PVLDB*, vol. 5, no. 10, 2012.

[11] M. N. Garofalakis, D. Keren, and V. Samoladas, "Sketch-based geometric monitoring of distributed stream queries," *PVLDB*, 2013.

[12] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *VLDB*, 2004.

[13] M. Tang, F. Li, J. M. Phillips, and J. Jestes, "Efficient threshold monitoring for distributed probabilistic data," in *ICDE*, 2012.

[14] B. Kanagal and A. Deshpande, "Online filtering, smoothing and probabilistic modeling of streaming data," in *ICDE*, 2008.

[15] J. Serra, "Image analysis and mathematical morphology," in *Academic Press, London*, 1982.

[16] S. Shah and K. Ramamritham, "Handling non-linear polynomial queries over dynamic data," in *ICDE*, 2008.

[17] D. Keren, D. B. Cooper, and J. Subrahmonia, "Describing complicated objects by implicit polynomials," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 1, 1994.

[18] E. Fogel and D. Halperin, "Exact and efficient construction of minkowski sums of convex polyhedra with applications," *Computer-Aided Design*, vol. 39, no. 11, 2007.

[19] M. Elad, A. Tal, and S. Ar, "Content based retrieval of vrml objects: an iterative and interactive approach," in *Proceedings of the sixth Eurographics workshop on Multimedia 2001*, 2002.

[20] R. Gupta, K. Ramamritham, and M. K. Mohania, "Ratio threshold queries over distributed data sources," in *ICDE*, 2010.

[21] "The european air quality database," in *http://tinyurl.com/ct9bh7x*.

[22] M. Kurpius and A. Goldstein, "Gas-phase chemistry dominates o3 loss to a forest, implying a source of aerosols and hydroxyl radicals to the atmosphere," *Geophysical Research Letters*, vol. 30, no. 7, 2007.

[23] http://tinyurl.com/kxssfgl.

[24] DCPR (Data Clustering and Pattern Recognition) Toolbox, http://tinyurl.com/nxospq2.

Communication-Efficient Distributed Online Prediction using Dynamic Model Synchronizations

[Extended Abstract]

Mario Boley and Michael Kamp
Fraunhofer IAIS & University Bonn
{mario.boley,michael.kamp}@iais.fraunhofer.de

Daniel Keren
Haifa University
dkeren@cs.haifa.ac.il

Assaf Schuster and Izchak Sharfman
Technion, Israel Institute of Technology
assaf@technion.ac.il & tsachis@technion.ac.il

ABSTRACT

We present the first protocol for distributed online prediction that aims to minimize online prediction loss and network communication at the same time. Applications include social content recommendation, algorithmic trading, and other scenarios where a configuration of local prediction models of high-frequency streams is used to provide a real-time service. For stationary data, the proposed protocol retains the asymptotic optimal regret of previous algorithms. At the same time, it allows to substantially reduce network communication, and, in contrast to previous approaches, it remains applicable when the data is non-stationary and shows rapid concept drift. The protocol is based on controlling the divergence of the local models in a decentralized way. Its beneficial properties are also confirmed empirically.

1. INTRODUCTION

We consider online prediction problems where data points are observed at local nodes in a distributed environment and there is a trade-off between maximizing prediction accuracy and minimizing network communication. This situation abounds in a wide range of machine learning applications, in which communication induces a severe cost. Examples are parallel data mining [Zinkevich et al., 2009, Hsu et al.] where communication constitutes a performance bottleneck, learning with mobile sensors [Nguyen et al., 2004, Predd et al., 2006] where communication drains battery power, and, most centrally, prediction-based real-time services [Dekel et al., 2012] carried out by several servers, e.g., for social content promotion, ad placement, or algorithmic trading. In addition to the above, here the cost of communication can also be a loss of prediction quality itself when training examples have to be discarded due to network latency.

1.1 Setting and Related Work

Despite the communication costs it induces, decentralization is inevitable in many modern scale applications. Hence, recent articles [Balcan et al., 2012, Daumé III et al.] explicitly investigate the communication complexity of learning with decentralized data. They consider, however, the *offline* task of finding a good global model over the union of all data as a final computation result. The same applies to some work on parallel machine learning (e.g., Zinkevich et al. [2010], McDonald et al. [2010]) where data shards are distributed among several processors and then all computation is carried out independently in parallel except for one final model merging step. While these approaches avoid communication for performance reasons, they do not intend to optimize the predictive performance during the computation. In contrast, we are interested in the *online* in-place performance, i.e., for every data point performance is assessed locally when and where it is received or sampled.

To this end, research focused so far on specific environments with fixed communication constraints. Correspondingly, the learning strategies that are proposed and analyzed for these settings, do not aim to minimize communication beyond the level that is enforced by these constraints. Zinkevich et al. [2009] considers a shared-memory model, in which all local nodes can update a global model in a round-robin fashion as they process their training examples. Since this approach is problematic if there is a notable communication latency, strategies have been investigated [Mann et al., 2009, Dekel et al., 2012] that communicate only periodically after a statically fixed number of data points have been processed. Dekel et al. [2012] shows that for smooth loss functions and stationary environments optimal asymptotic regret bounds can be retained by updating a global model only after *mini-batches* of $O(\sqrt[3]{m})$ data points. Here, m denotes the total number of data points observed throughout the lifetime of the system. For large values of m, the effect of bounded latency values is asymptotically outgrown by the increasing mini-batch size.

While a fixed periodic communication schedule reduces the communication by some fixed amount, further reduction is desirable: The above mentioned costs of communication can have a severe impact on the practical performance—even if they are not reflected in asymptotic performance bounds. This is further amplified because a large number of modeling

tasks are performed simultaneously sharing the same limited bandwidth. Moreover, distributed learning systems that are deployed for a long lifetime relative to their data throughput can experience periodical or singular target drifts (e.g., corresponding to micro-trends in social networks). In these settings, a static schedule is bound to either provide only little to no communication reduction or to insufficiently react to changing data distributions.

1.2 Contributions and Outline

In this work, we give the first distributed prediction protocol for linear models that, at the same time, aims to provide a high online in-place prediction performance and explicitly tries to minimize communication. In terms of predictive power, as shown Sec. 3.1, the protocol retains the asymptotic optimal regret of the distributed mini-batch algorithm of Dekel et al. [2012] for stationary data. In addition, it allows to reduce the communication among the local nodes substantially. This is achieved by a dynamic data dependent communication schedule, which, in contrast to previous algorithms, remains applicable when the data is non-stationary and shows rapid concept drifts. The main idea is to synchronize the local models to their mean model in order to reduce their variance, but to do so only in system states that show a high divergence among the models. This divergence, measured by the average model distance to the mean model, indicates the synchronizations that are most important in terms of their correcting effects on the predictions. In stable phases this allows communicative quiescence, while, in hard phases where variance reduction is crucial, the protocol will trigger a lot of model synchronizations. In order to efficiently implement this strategy one has to monitor the non-linear divergence function without communication overhead. We propose a solution to this problem that adapts recent ideas from distributed systems research based on local safe-zones in the function domain (Sec. 3.2). Experiments confirm the beneficial properties of the protocol (Sec. 4).

2. PRELIMINARIES

In this section we formally introduce the distributed online prediction task. As simple local learning tool we recall stochastic gradient descent for linear models. Finally, we review the state-of-the-art communication protocol as a departure point for developing a more communication-efficient solution in subsequent sections.

2.1 Distributed Online Prediction

Throughout this paper we consider a distributed online prediction system of k **local learners** that maintain individual **linear models** $w_{t,1}, \ldots, w_{t,k} \in \mathbb{R}^n$ of some global environment through discrete time $t \in [T]$ where $T \in \mathbb{N}$ denotes the total time horizon with respect to which we analyze the system's performance. This environment is represented by a **target distribution** $\mathcal{D}_t : X \times Y \to [0,1]$ that describes the relation between an input space $X \subseteq \mathbb{R}^n$ and an output space $Y \subseteq \mathbb{R}$. The nature of Y varies with the learning task at hand; $Y = \{-1, 1\}$ is used for binary classification, $Y = \mathbb{R}$ for regression. While we allow $\mathcal{D}_t$ to vary with time, we assume that it remains constant most of the time and only experiences a small number of rapid drifts. That is, there are **drift points** $0 = d_0 < d_1 < \cdots < d_p = T$ such that for all $i \in [p]$ and $t, t' \in [T]$ with $d_{i-1} \leq t \leq t' < d_i$ it holds that $\mathcal{D}_t = \mathcal{D}_{t'}$. Hence, there are identically distributed **episodes** $E_i = \{d_i, \ldots, d_{i+1} - 1\}$ between any two drift points. We assume that all learners sample from $\mathcal{D}$ independently in parallel using a constant and uniform sampling frequency, and we denote by $(x_{t,l}, y_{t,l}) \sim \mathcal{D}_t$ the **training example** received at node l at time t. Generally, we assume that all training examples are bounded by a ball with **radius** R.

Conceptually, every learner first observes the input part $x_{t,l}$ and performs a real time service based on the linear **prediction score** $p_{t,l} = \langle w_{t,l}, x_{t,l} \rangle$, i.e., the inner product of $x_{t,l}$ and the learner's current model vector. Only then it receives as feedback the true label $y_{t,l}$, which it can use to locally update its model to $w_{t+1,l} = \varphi(w_{t,l}, x_{t,l}, y_{t,l})$ by some **update rule** $\varphi : \mathbb{R}^n \times X \times Y \to \mathbb{R}^n$. Finally, the learners are connected by a communication infrastructure that allows them to jointly perform a **synchronization operation** $\sigma : \mathbb{R}^{k \times n} \to \mathbb{R}^{k \times n}$ that resets the whole model configuration to a new state and that may take into account the information of all local learners simultaneously. The performance of such a distributed online prediction system is measured by two quantities: 1) the predictive performance $\sum_{t=1}^{T} \sum_{l=1}^{k} f(p_{t,l}, y_{t,l})$ measured by a **loss function** $f : \mathbb{R} \times Y \to \mathbb{R}_+$ that assigns positive penalties to prediction scores; and 2) the amount of **communication** within the system that is measured by the number of bits sent inbetween learners to compute the sync operation. Next, specify possible choices for the update rule, the loss function, and the synchronization operator.

2.2 Losses and Gradient Descent

Generally, the communication protocol developed in this paper is applicable to a wide range of online update rules for linear models from, e.g., the passive aggressive rule [Crammer and Singer, 2001] to regularized dual averaging [Xiao, 2010]. However, the regret bound given in Theorem 2 assumes that the updates are **contractions**. That is, there is some constant $c < 1$ such that for all $w, w' \in \mathbb{R}^n$, and $x, y \in X \times Y$ it holds that $\|\varphi(w, x, y) - \varphi(w', x, y)\| \leq c\|w - w'\|$. For the sake of simplicity, in this paper, we focus on rules based on l_2-regularized stochastic gradient descent, for which this contraction property is readily available. We note that by considering *expected* contractions the result can be extended to rules that reduce on average the distance to a (regularized) loss minimizer.

Before we can define gradient descent updates, we have to introduce the underlying loss functions measuring predictive performance. Again for convenience, we restrict ourselves to functions that are differentiable, convex, and globally **Lipschitz continuous** in the prediction score, i.e., there is some constant L such that for all $p, p', y \in \mathbb{R}^{2n} \times Y$ it holds that $|f(p, y) - f(p', y)| \leq L|p - p'|$. While these assumptions can be relaxed by spending some technical effort, they already include loss functions for all standard predictions tasks such as the **logistic loss** $f_{\text{lg}}(p, y) = \ln(1 + \exp(-yp))$ for binary classification (case $Y = \{-1, 1\}$) or the **Huber loss** for regression (in the case $Y = \mathbb{R}$)

$$f_{\text{hu}}(p, y) = \begin{cases} \frac{1}{2}(p - y)^2 & , \text{ for } |p - y| \leq 1 \\ |p - y| - \frac{1}{2} \end{cases} .$$

See, e.g., Zhang [2004] for further possible choices. In both of these cases the (best) Lipschitz constant is $L = 1$.

Algorithm 1 Static Synchronization Protocol

Initialization:
  local models $w_{1,1}, \ldots, w_{1,k} \leftarrow (0, \ldots, 0)$

Round t at node l:
  observe $x_{t,l}$ and provide service based on $p_{t,l}$
  observe $y_{t,l}$ and **update** $w_{t+1,l} \leftarrow \varphi(w_{t,l}, x_t, y_t)$
  if $t \mod b = 0$ **then**
    send $w_{t,l}$ to coordinator

At coordinator every b rounds:
  receive local models $\{w_{t,l} \colon l \in [k]\}$
  send $w_{t,1}, \ldots, w_{t,k} \leftarrow \frac{1}{k} \sum_{l \in [k]} w_l$

With this we can define **stochastic gradient descent (SGD)** rules with l_2-regularization, i.e., rules of the form

$$\varphi(w, x, y) = w - \eta_t \nabla_w \left(\frac{\lambda}{2} \|w\|^2 + f(\langle w, x \rangle, y) \right)$$

where $\lambda \in \mathbb{R}_+$ is a strictly positive **regularization parameter** and $\eta_t \in \mathbb{R}_+$ are strictly positive **learning rates** for $t \in \mathbb{N}$. For stationary target distributions, one often chooses a decreasing learning rate such as $\eta_t = 1/\sqrt{t}$ in order to guarantee convergence of the learning process. For non-stationary targets this is infeasible, because for large t it would prevent sufficient model adaption to target changes. However, one can show [Zinkevich et al., 2010] that stochastic gradient descent is a contraction for sufficiently small constant learning rates. Namely, for $\eta \leq (RL + \lambda)^{-1}$ the updates do contract with constant $c = 1 - \eta\lambda$. This can be used to show that the stochastic learning process converges to a distribution centered close to a regularized loss minimizer even when the process is distributed among k nodes (see the analysis of Zinkevich et al. [2010]). This refers to the stochastic learning process defined by the mean of independent local models that result from SGD with iid samples from (episodes of) the target distribution. In this paper, the contraction property is used for the regret bound of Thm. 2.

2.3 Communication and Mini-batches

For every episode E_i, the predictive performance of a distributed prediction system lies between two baselines that correspond to the two extremes in terms of communication behavior—complete centralization and no communication. Let $T_i = |E_i|$ denote the length of episode E_i and by $R = \sum_{t \in E_i,\, l \in [k]} f(p_{t,l}, y_{t,l}) - f^*$ the regret with respect to the optimal expected loss $f^* = \operatorname{argmin}_{w \in \mathbb{R}^n} \mathbb{E}_{(x,y) \sim \mathcal{D}_i}[f(\langle w, x \rangle, y)]$. When all data points are centrally processed by one online learner, for long enough episodes one can achieve an expected regret of $O(\sqrt{kT_i})$ which is optimal (see Cesa-Bianchi and Lugosi [2006] and Abernethy et al. [2009]). In contrast, when the k nodes perform their learning processes in parallel without any communication this results in an expected regret of $O(k\sqrt{T_i})$, which is worse than the centralized performance by a factor of $\sqrt{k}$. Therefore, we are interested in algorithms that lie between these two extremes and that show a beneficial trade-off between predictive performance and the amount communication.

Mann et al. [2009] and Dekel et al. [2012] give algorithms where information between nodes is only exchanged every b rounds where $b \in \mathbb{N}$ is referred to as **batch size**. These algorithms can be written as static model synchronization protocol similar to Alg. 1. Here, after a batch of kb examples has been processed globally in the system, all local models are re-set to the **mean model** of the configuration $\mathbf{w}$ defined as $\overline{\mathbf{w}} = 1/k \sum_{l=1}^{k} w_l$. Formally, the synchronization operator that is implicitly employed in these algorithms is given by $\sigma(\mathbf{w}_t) = (\overline{\mathbf{w}}_t, \ldots, \overline{\mathbf{w}}_t)$. We refer to this operation as **full mean synchronization**. The choice of a (uniform) model mixture is often used for combining linear models that have been learned in parallel on independent training data (see Mann et al. [2009], McDonald et al. [2010], Zinkevich et al. [2010]). The motivation is that the mean of k models provides a variance reduction of $\sqrt{k}$ over an individual random model (recall that all learners sample from the same distribution, hence their models are identically distributed). Dekel et al. [2012] shows that when the gradient variance is bounded then the optimal regret can be asymptotically retained by setting $b = O(\sqrt[3]{T_i})$ even if a constant number of examples have to be discarded during each synchronization due to network latency. Note that this reference considers a slightly modified algorithm based on delayed gradient descent, which only applies (accumulated) updates at synchronization points. However, the expected loss of eager updates (as used in Alg. 1) is bounded by the expected loss of delayed updates (as used in Dekel et al. [2012]) as long as the updates reduce the distance to a loss minimizer on average (which is the case for sufficiently small learning rates and regularization parameters; see again Zhang [2004, Eq. 5]).

Closing this section, let us analyze the communication cost of this protocol.Using a designated coordinator note as in Alg. 1, σ can computed simply by all nodes sending their current model to the coordinator, who in turn computes the mean model and sends it back to all the nodes. For assessing the communication cost of this operation, we only count the number of model vectors sent between the learners. This is feasible because, independently of the exact communication infrastructure, the number of model messages asymptotically determines the true bit-based cost. Hence, asymptotically the **communication cost** of static model synchronization over k nodes with batch size b is $O(kT/b)$. Dekel et al. [2012] assumes that the data distribution is stationary over all rounds and b can therefore be set to $O(\sqrt[3]{T})$. This results in an automatic communication reduction that increases with a longer system lifetime. However, this strategy is not applicable when we want to stay adaptive towards changing data distributions. In this case, we have to set the batch size with respect to the expected episode length and not with respect to the overall system lifetime. This number can be much smaller than T resulting in batch sizes that are too small to meet our communication reduction goal. In the following section, we therefore design a synchronization protocol that can substantially reduce this cost based on a data-dependent dynamic schedule.

3. DYNAMIC SYNCHRONIZATION

The synchronization protocol of Alg. 1 is static because it synchronizes after a fixed number of rounds independently of the sampled data and its effect on the local models. Consequently, it incurs the communication cost of a full synchronization round even if the models are (almost) identical and thus only receive little to none correction. In this section, we develop a dynamic protocol for synchronizations based on quantifying their effect. After showing that this approach

is sound from a learning perspective, we discuss how it can be implemented in a communication-efficient way.

3.1 Partial Synchronizations

A simple measure to quantify the correcting effect of synchronizations is given by the average Euclidean distance between the current local models and the result model. We refer to this quantity as the **divergence** of a model configuration, denoted by $\delta(\cdot)$, i.e., $\delta(\mathbf{w}) = \frac{1}{k}\sum_{l=1}^{k} \|\overline{\mathbf{w}} - \mathbf{w}_l\|_2$. In the following definition we provide a relaxation of the full mean synchronization operation that introduces some leeway in terms of this divergence.

DEFINITION 1. *A **partial synchronization operator** with a positive divergence threshold $\Delta \in \mathbb{R}$ is an operator $\sigma_\Delta : \mathbb{R}^{k\times n} \to \mathbb{R}^{k\times n}$ that 1) leaves the mean model invariant and 2) after its application the model divergence is bounded by Δ. That is, for all model configurations $\mathbf{w} \in \mathbb{R}^{k\times n}$ it holds that $\overline{\mathbf{w}} = \overline{\sigma_\Delta \mathbf{w}}$ and $\delta(\sigma_\Delta \mathbf{w}) \leq \Delta$.*

An operator adhering to this definition does not generally put all nodes into sync (albeit the fact that we still refer to it as *synchronization* operator). In particular it allows to leave all models untouched as long as the divergence remains below the threshold Δ. The following theorem notes that partial synchronization has a controlled regret over full synchronization if the batch size is sufficiently large and the divergence threshold is set proportional to the Lipschitz constant L of the losses and the data radius R.

THEOREM 2. *Suppose the update rule φ is a contraction with constant c. Then, for batch sizes $b \geq \log_2^{-1} c^{-1}$ and divergence thresholds $\Delta \leq \epsilon/(2RL)$, the average regret of using a partial synchronization operator σ_Δ instead of σ is bounded by ϵ, i.e., for all rounds $t \in \mathbb{N}$ it holds that the average regret $1/k\sum_{l=1}^{k} |f(p_{t,l}^{\Delta}, y_{t,l}) - f(p_{t,l}, y_{t,l})|$ is bounded by ϵ where $p_{t,l}$ and $p_{t,l}^{\Delta}$ denote the prediction scores at learner l and time t resulting from σ and σ_Δ, respectively.*

We omit the proof here referring to the full version of this paper. While the contraction assumption is readily available for regularized SGD, as mentioned in Sec. 2, it can be relaxed: by requiring the updates to only contract on *expectation* it is possible to extend the theorem to unregularized SGD updates as well as to other rules. Moreover, we remark that Thm. 2 implies that partial synchronizations retain the optimality of the static mini-batch algorithm of Dekel et al. [2012] for the case of stationary targets: By using a time-dependent divergence threshold based on $\epsilon_t \in O(1/\sqrt{t})$ the bound of $O(\sqrt{T})$ follows.

3.2 Communication-efficient Protocol

After seeing that partial synchronization operators are sound from the learning perspective, we now turn to how they can be implemented in a communication-efficient way. Every distributed learning protocol that implements a partial synchronization operator has to implicitly control the divergence of the model configuration. However, we cannot simply compute the divergence by centralizing all local models, because this would incur just as much communication as static full synchronization. Our strategy to overcome this problem is to first decompose the global condition $\delta(\mathbf{w}) \leq \Delta$ into a set of local conditions that can be monitored at their respective nodes without communication (see, e.g., Sharfman et al. [2007]). Secondly, we define a resolution protocol

Algorithm 2 Dynamic Synchronization Protocol

Initialization:
  local models $w_{1,1}, \ldots, w_{1,k} \leftarrow (0, \ldots, 0)$
  reference point $r \leftarrow (0, \ldots, 0)$
  violation counter $v \leftarrow 0$

Round t at node l:
  observe $x_{t,l}$ and provide service based on $p_{t,l}$
  observe $y_{t,l}$ and **update** $w_{t+1,l} \leftarrow \varphi(w_{t,l}, x_t, y_t)$
  if $t \mod b = 0$ **and** $\|r - w_{t,l}\| > \Delta/2$ **then**
    send $w_{t,l}$ to coordinator

At coordinator on violation:
  let B be set of nodes with violation
  $v \leftarrow v + |B|$
  if $v = k$ **then** $B \leftarrow [k]$, $v \leftarrow 0$
  while $B \neq [k]$ **and** $\|r - \frac{1}{B}\sum_{l\in B} w_l\| > \Delta$ **do**
    augment B by augmentation strategy
    receive models from nodes added to B
  send to nodes in B model $w = \frac{1}{B}\sum_{l\in B} w_l$
  if $B = [k]$ also set new reference model $r \leftarrow w$

that transfers the system back into a valid state whenever one or more of these local conditions are violated. This includes carrying out a sufficient amount of synchronization to reduce the divergence to be less or equal than Δ.

For deriving local conditions we consider the domain of the divergence function restricted to an individual model vector. Here, we identify a *safe-zone* S (see Keren et al. [2012]) such that the global divergence can not cross the Δ-threshold as long as all local models remain in S.[1] The following statement, which we give again without proof, provides a valid spherical safe zone S_r that is centered around some global reference point r.

THEOREM 3. *Let $r \in \mathbb{R}^d$ be some reference point. If for all nodes $l \in \{1, \ldots, k\}$ it holds that $\|r - w_l\| \leq \Delta/2$ then we have for the model divergence that $\delta(\mathbf{w}) \leq \Delta$.*

We now incorporate these local conditions into a distributed prediction protocol. As a first step, we have to guarantee that at all times all nodes use the same reference point. For a prediction t, let us denote by t' the last time prior to t when a full model synchronization was performed (resp. $t' = 0$ in case no full synchronization has happened until round t). The mean model $\overline{\mathbf{w}}_{t'}$ is known to all local learners. We use this model as the reference model and set $r = \overline{\mathbf{w}}_{t'}$. A local learners l can then monitor their local condition $\|r - w_l\| \leq \Delta/2$ in a decentralized manner.

It remains to design a resolution protocol that specifies how to react when one or several of the local conditions are violated. A direct solution is to trigger a full synchronization in that case. This approach, however, does not scale well with a high number of nodes in cases where model updates have a non-zero probability even in the asymptotic regime of the learning process. When, e.g., PAC models for the current target distribution are present at all local nodes, the probability of one local violation, albeit very low for an individual node, increases exponentially with the number of nodes. An alternative approach that can keep the amount

[1] Note that a direct distribution of the threshold across the local nodes (as in, e.g., Keralapura et al. [2006]) is infeasible, because the divergence function is non-linear.

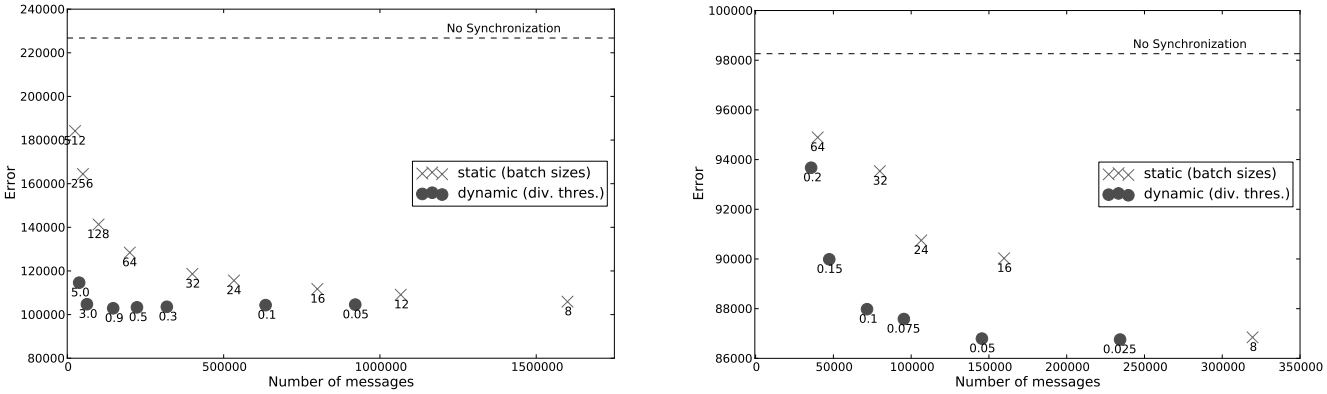


Figure 1: Performance of static and dynamic model synchronization that track (left) a rapidly drifting disjunction over 100-dimensional data with 512 nodes; and (right) a neural network with one hidden layer and 150 output variables. with 1024 nodes.

of communication low relative to the number of nodes is to perform a local balancing procedure: on a violation, the respective node sends his model to a designated note we refer to as coordinator. The coordinator then tries to balance this violation by incrementally querying other nodes for their models. If the mean of all received models lies within the safe zone, it is transferred back as new model to all participating nodes, and the resolution is finished. If all nodes have been queried, the result is equal to a full synchronization and the reference point can be updated. In both cases, the divergence of the model configuration is bounded by Δ at the end of the balancing process, because all local conditions hold. Also this protocol leaves the global mean model unchanged. Hence, it is complying to Def. 1.

While balancing can achieve a high communication reduction over direct resolution particularly for a large number of nodes, it potentially degenerates in certain special situations: We can end up in a stable regime in which local violations are likely to be balanced by a subset of the nodes; however a full synchronization would strongly reduce the expected number of violations in future rounds. In other words: balancing can delay crucial reference point updates indefinitely. A simple hedging mechanism for online optimization can be employed to avoid this situation: we count the number of local violations using the current reference point and trigger a full synchronization whenever this number exceeds the number of nodes. This concludes our dynamic protocol for distributed prediction. All components are summarized in Alg. 2

4. EMPIRICAL EVALUATION

In this section we investigate the practical performance of the dynamic learning protocol for two controlled settings: one with linearly separable data and one with unseparable data. Our main goal is to empirically confirm that the predictive gain of static full synchronizations (using a batch size of 8) over no synchronization can be approximately preserved for small enough thresholds, and to assess the amount of communication reduction achieved by these thresholds.

We start with the problem of tracking a rapidly drifting random disjunction. In this case the target distribution produces data that is episode-wise linearly separable. Hence, we can set up the individual learning processes so that they converge to a linear model with zero classification error within each episode. Formally, we identify a target disjunction with a binary vector $z \in \{0,1\}^n$. A data point $x \in X = \{0,1\}^n$ is labeled positively $y = 1$ if $\langle x, z \rangle \geq 1$ and otherwise receives a negative label $y = -1$. The target disjunction is drawn randomly at the beginning of the learning process and is randomly re-set after each round with a fixed drift probability of 0.0002. In order to have balanced classes, the disjunctions as well as the data points are generated such that each coordinate is set independently to 1 with probability $\sqrt{1 - 2^{-1/n}}$. As loss function for the stochastic gradient descent we use the logistic loss. Corresponding to our setting of noise-free linearly separable data, we choose the regularization parameter $\lambda = 0$ and the learning rate $\eta = 1$.

In Fig. 1 (left) we present the result for dimensionality $n = 100$, with $k = 512$ nodes, processing $m = 12.8M$ data points through $T = 25000$ rounds. For divergence thresholds up to 3.0, dynamic synchronization can retain the error number of statically synchronizing every 8 rounds. At the same time the communication is reduced to 3.9% of the original number of messages. An approximately similar amount of communication reduction can also be achieved using static synchronization by increasing the batch size to 128. This approach, however, only retains 51.5% of the error reduction over no communication. Analyzing the development of the evaluation metrics over time reveals: At the beginning of each episode there is a relatively short phase in which additional errors are accumulated and the communicative protocols acquire an advantage over the baseline of never synchronizing. This is followed by a phase during which no additional error is made. Here, the communication curve of the dynamic protocols remain constant acquiring a gain over the static protocols in terms of communication.

We now turn to a harder experimental setting, in which the target distribution is given by a rapidly drifting two-layer neural network. For this target even the Bayes optimal classifier per episode has a non-zero error, and, in particular, the generated data is not linearly separable. Intuitively, it is harder in this setting to save communication, because a non-zero residual error can cause the linear models to periodically fluctuate around a local loss minimizer—resulting in crossings of the divergence threshold even when the learning processes have reached their asymptotic regime. We choose the network structure and parameter ranges in

a way that allow for a relatively good approximation by linear models (see Bshouty and Long [2012]). The process for generating a single labeled data point is as follows: First, the label $y \in Y = \{-1, 1\}$ is drawn uniformly from Y. Then, values are determined for hidden variables H_i with $1 \leq i \leq \lceil \log n \rceil$ based on a Bernoulli distribution $P[H_i = \cdot | Y = y] = \text{Ber}(p^h_{i,y})$. Finally, $x \in X = \{-1, 1\}^n$ is determined by drawing x_i for $1 \leq i \leq n$ according to $P[X_i = x_i, |H_{p(i)} = h] = \text{Ber}(p^o_{i,h})$ where $p(i)$ denotes the unique hidden layer parent of x_i. In order to ensure linear approximability, the parameters of the output layer are drawn such that $|p^o_{i,-1} - p^o_{i,1}| \geq 0.9$, i.e., their values have a high *relevance* in determining the hidden values. As in the disjunction case all parameters are re-set randomly after each round with a fixed drift probability (here, 0.005). For this non-separable setting we choose again to optimize the logistic loss, this time with parameters $\lambda = 0.5$ and $\eta = 0.05$ respectively. Also, in order to increase the stability of the learning process, we apply averaged updates over mini-batches of size 8.

Figure 1 (right) contains the results for dimensionality 150, with $k = 1024$ nodes, processing $m = 2.56M$ data points through $T = 2500$ rounds. For divergence thresholds up to 0.05, dynamic synchronization can retain the error of the baseline. At the same time the communication is reduced to 46% of the original number of messages.

5. CONCLUSION

We presented a protocol for distributed online prediction that aims to dynamically save on network communications in sufficiently easy phases of the modeling task. The protocol has a controlled predictive regret over its static counterpart and experiments show that it can indeed reduce the communication substantially—up to 95% in settings where the linear learning processes are suitable to model the data well and converge reasonably fast. Generally, the effectivity of the approach appears to correspond to the effectivity of linear modeling by SGD in the given setting.

For future research a theoretical characterization of this behavior is desirable. A practically even more important direction is to extend the approach to other model classes that can tackle a wider range of learning problems. In principle, the approach of controlling model divergence remains applicable, as long as the divergence is measured with respect to a distance function that induces a useful loss bound between two models. For probabilistic models this can for instance be the KL-divergence. However, more complex distance functions constitute more challenging distributed monitoring tasks, which currently are open problems.

References

Jacob Abernethy, Alekh Agarwal, Peter L. Bartlett, and Alexander Rakhlin. A stochastic view of optimal regret through minimax duality. In *COLT 2009 - The 22nd Conference on Learning Theory*, 2009.

Maria-Florina Balcan, Avrim Blum, Shai Fine, and Yishay Mansour. Distributed learning, communication complexity and privacy. *Journal of Machine Learning Research - Proceedings Track*, 23:26.1–26.22, 2012.

Nader H. Bshouty and Philip M. Long. Linear classifiers are nearly optimal when hidden variables have diverse effects. *Machine Learning*, 86(2):209–231, 2012.

Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006. ISBN 978-0-521-84108-5.

Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:265–292, 2001.

Hal Daumé III, Jeff M. Phillips, Avishek Saha, and Suresh Venkatasubramanian. Efficient protocols for distributed classification and optimization. In *ALT 2012*.

Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13:165–202, 2012.

Daniel Hsu, Nikos Karampatziakis, John Langford, and Alexander J. Smola. Parallel online learning. In *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press.

Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD 2006)*, pages 289–300, 2006.

Daniel Keren, Izchak Sharfman, Assaf Schuster, and Avishay Livne. Shape sensitive geometric monitoring. *Knowledge and Data Engineering, IEEE Transactions on*, 24(8):1520–1535, 2012.

G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems (NIPS 2009)*, volume 22, pages 1231–1239, 2009.

Ryan T. McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Human Language Technologies: Conf. of the North American Chapter of the Association of Computational Linguistics, Proceedings (HLT-NAACL)*, pages 456–464, 2010.

XuanLong Nguyen, Martin J Wainwright, and Michael I Jordan. Decentralized detection and classification using kernel methods. In *Proceedings of the twenty-first international conference on Machine learning*, page 80. ACM, 2004.

Joel B Predd, SB Kulkarni, and H Vincent Poor. Distributed learning in wireless sensor networks. *Signal Processing Magazine, IEEE*, 23(4):56–69, 2006.

Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Trans. Database Syst.*, 32 (4), 2007.

Lin Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *The Journal of Machine Learning Research*, 11:2543–2596, 2010.

Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the 21st int. conf. on Machine learning (ICML 2004)*, 2004.

Martin Zinkevich, Alex J. Smola, and John Langford. Slow learners are fast. In *Proc. of 23rd Annual Conference on Neural Information Processing Systems (NIPS 2009)*, pages 2331–2339, 2009.

Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *Proc. of 24th Annual Conference on Neural Information Processing Systems (NIPS 2010)*, pages 2595–2603, 2010.

Communication-efficient Outlier Detection for Scale-out Systems

Moshe Gabel
Technion
Haifa, Israel
mgabel@cs.technion.ac.il

Daniel Keren
Haifa University
Haifa, Israel
dkeren@cs.haifa.ac.il

Assaf Schuster
Technion
Haifa, Israel
assaf@cs.technion.ac.il

ABSTRACT

Modern scale-out services are built on top of large datacenters composed of thousands of individual machines. These must be continuously monitored because unexpected failures can overload fail-over mechanism and cause large-scale outages. Such monitoring can be accomplished by periodically measuring hundreds of performance metrics and looking for outliers, often caused by misconfigurations, hardware failures or even software bugs. Previous work has shown that many failures are indeed preceded by such performance outliers, known as *performance problems* or *latent faults*.

In this work we adapt an existing unsupervised statistical framework for latent fault detection to provide an online, communication- and computation-reduced version. The existing framework is effective in predicting machine failures days before they happen, but requires each monitored machine to send all its periodic metric measurements, which is prohibitive in some settings and requires that the datacenter provide parallel storage and processing. Our adapted framework is able to reduce the amount of data sent and the processing cost at the central coordinator by processing the data in situ, making it usable in wider settings.

We utilize techniques from the domain of stream processing, specifically *sketching* and *safe zones*, to trade-off accuracy for communication and computation, without compromising its advantages. Like the original framework, our adapted framework is unsupervised, does not require domain knowledge, and provides statistical guarantees on the rate of false positives. Initial experiments show that scores yielded by the adapted framework match the original scores very well, while reducing communications by over 90%.

1. INTRODUCTION

In recent years the demand for computing power and storage has increased. Modern Web services and clouds rely on large datacenters, often comprised of thousands of machines. For such large services, it is unreasonable to assume that all machines are working properly and are well configured.

Monitoring is essential in datacenters, since unnoticed faults might accumulate to the point where redundancy and fail-over mechanisms break. Yet the large number of machines in datacenters makes manual monitoring impractical. Instead machines are usually monitored by collecting and analyzing performance counters [3, 5, 11]. Hundreds of counters per machine are reported by the various service layers, from service-specific metrics (such as database query statistics) to general metrics (such as CPU utilization).

In this work we adapt an existing fault detection algorithm [9] using sketching [8, 18, 7] and safe zones [17, 21] to reduce communication and processing requirements by an order of magnitude, while preserving its advantages.

Many existing failure detectors are inflexible [9], and most require centralizing the data in some form. Rule-based failure detectors define a set of watchdogs [11] that monitor specific counters and trigger an alert whenever a predefined threshold is crossed. However, maintaining these static rules requires ongoing manual adjustments.

More advanced methods model service behavior from historical logs. Supervised machine learning approaches [3, 6, 20, 4] train detectors on historic annotated data. Others [5] analyze logs from periods from when the service is guaranteed to be healthy to extract model parameters. Such approaches are sensitive to deviations in workloads and changes in the monitored service itself [23, 10]. After such changes the historical logs and the learned model are no longer relevant. Approaches that require labeled data can be expensive, since labels can be difficult to obtain, and re-labeling may be needed after service changes.

More flexible, unsupervised approaches have been proposed for high performance computing (HPC). Typical approaches [19, 22] analyze textual console logs to detect system or machine failures by examining frequency of log messages. Console logs are impractical in high-volume services for bandwidth and performance reasons: transactions are very short, time-sensitive, and rapid.

Finally, some approaches [14, 16] are unsupervised and flexible, but are not domain independent. They make use of domain insights and knowledge of the monitored service, for example in the domain of distributed file systems, and are therefore limited to specific systems.

Recent approaches to the monitoring problem [9, 16, 15] focus on early detection and handling of performance problems, or *latent faults*. These are outliers – machine behaviors that are indicative of a fault, or could eventually result in a fault, yet fly under the radar of monitoring systems because they are not acute enough, or were not anticipated by the

monitoring system designers. Early detection of latent faults can help prevent future failures and increase the reliability of services.

In previous work [9] we provided evidence that latent faults are common, and we presented a novel, unsupervised outlier detection framework for latent fault detection. In experiments on a real-world production system comprised of 4500 machines, we showed that over 20% of machine failures were preceded by latent faults. Furthermore, we were able to detect latent faults up to 14 days in advance of actual machine or software failures with up to 70% precision and 2% false positive rate – comparable to state of the art supervised techniques in controlled settings [4]. We demonstrated that our system is adaptable, requiring no domain knowledge, no labeled examples, and no parameter tuning in the face of workload changes and software updates. Finally, our system has proven and demonstrated guarantees on the false positive rates, it is non-intrusive, and it scales to very large services.

One drawback of previous work is the large communication and processing costs, prohibitive in some settings. Modern data centers are large, and consequently the resultant counter logs are also large. It may be very difficult to centralize and process such a large amount of data. In the experiments described in [9], the log files were over 10TB per day – too large to centralize and process in one location. Instead we relied on a data-parallel infrastructure [12] built into the data center. Parallel processing may not always be feasible in all situations, however. Furthermore, some large systems are not confined to a single datacenter but are geographically distributed.

In this work we extend our latent fault detection using techniques from the field of stream processing to reduce the size of the data by an order of magnitude, reducing communication and processing requirements, and allowing continuous online processing of distributed streams. The resulting technique is essentially a distributed outlier detector for multiple multivariate data streams, designed for monitoring large-scale online services.

2. SUMMARY OF PREVIOUS WORK

In [9] we presented a statistical latent fault detection framework with 3 derived tests. What follows is a short summary of that work, with the sign test as example.

2.1 Framework

We begin with a reasonable assumption: in a large cluster of machines doing the same job, most machines perform well most of the time. Further, we expect similar machines with similar hardware and software[1] to exhibit roughly similar behavior when measuring performance counters. We therefore compare these machines to find those whose performance differs notably.

There are M machines, each reporting C performance counters at every time t in a window of length T time points. We denote by $x(m,t)$ the vector of counter values for machine m at time t. The hypothesis is that the inspected machine is working properly and hence the statistical process that generated this vector for machine m is the same statistical process that generated the vector for any other machine m'. However, if we see that the vector $x(m,t)$ for machine m is notably different from the vectors of other machines, we reject the hypothesis and flag the machine m as *suspicious*, meaning we suspect it manifests a latent fault.

We now make explicit our assumptions on the behavior of the monitored machines: *a*) the majority of machines are working properly at any given point in time; *b*) the machines are homogeneous, meaning they perform a similar task and use similar hardware and software[2]; *c*) on average, the workload is balanced across all machines; *d*) the counters are ordinal and are reported at the same rate; and *e*) the counter values are memoryless in the sense that they depend only on the current time period (and are independent of the identity of the machine).

Formally, we assume that $x(m,t)$ is a realization of a random variable $X(t)$ whenever machine m is working properly. Since all machines perform the same task, and since the load balancer attempts to split the load evenly between the machines, the homogeneous assumption implies that we should expect $x(m,t)$ to show similar behavior. We do expect to see changes over time, due to changes in the workload, for example. However, we expect these changes to be similarly reflected in all machines.

At any time t, the input $x(t)$ to a test S consists of the vectors $x(m,t)$ for all machines m. The test $S(m,x(t))$ analyzes the data and assigns a *score* (either a scalar or a vector) to machine m at time t. Given a test S, and a significance level $\alpha > 0$, we can present the framework as follows:

1. Preprocess: select counters and scale to unit variance;
2. Compute for every machine m the vector: $v_m = \frac{1}{T}\sum_t S(m,x(t))$ (integration phase);
3. Compute the p-values (defined below) $p(m)$ from v_m;
4. Report every machine with $p(m) < \alpha$ as suspicious.

Essentially, the scores for machine m are aggregated over time, so that eventually the norm of the aggregated scores converges, and is used to compute a p-value for m. The longer the allowed time period for aggregating the scores is, the more sensitive the test will be. At the same time, aggregating over long periods of time creates latencies in the detection process. In our previous work we aggregated data over 24 hour intervals, as a compromise between sensitivity and latency.

The p-value for a machine m is a bound on the probability that a random healthy machine would exhibit such aberrant counter values. If the p-value falls below a predefined significance level α, the null hypothesis is rejected, and the machine is flagged as suspicious.

In [9] we derived and evaluated 3 different tests within the framework (different S functions). The sign test accumulates the average normalized direction from machine m to the rest of the machines. The Tukey test measures the average depth of $x(m,t)$ compared to the vectors of other machines at the same time. The LOF test similarly compares the local density of points around $x(m,t)$ to the local density of its neighbors. What follows is a summary of the sign test.

[1]These are reasonable assumptions in practice for many services and datacenters [14, 19].

[2]If this is not the case, we can often split the collection of machines to a few large homogeneous clusters.)

2.2 The Sign Test

The sign test extends the classic statistical sign test to allow the simultaneous comparison of multiple machines. The "sign" of a machine m at time t is the average direction of its vector $x(m,t)$ to all other machines' vectors, and its score v_m is the sum of all these directions, divided by T.

The intuition is that healthy machines are similar on average, and any differences are random. Average directions are therefore random and tend to cancel each other out when added together, meaning v_m will be a relatively short vector for healthy machines. Conversely, if m has a latent fault, then some of its metrics are consistently different from healthy machines, and so the average directions are similar in some dimensions. When summing up these average directions, these similarities reinforce each other and therefore v_m tends to be a longer vector.

Formally, let $\mathcal{M}$ denote the set of all machines in a test, and $M = |\mathcal{M}|$ the number of machines. $\mathcal{T}$ are the time points where counters are sampled during preprocessing (for instance, every 5 minutes for 24 hours in our experiments), t denote a specific time point, and $T = |\mathcal{T}|$. Let m and m' be two machines and let $x(m,t)$ and $x(m',t)$ be the vectors of their reported and preprocessed counters at time t. We use the test

$$S\left(m, x(t)\right) = \frac{1}{M-1} \sum_{m' \neq m} \frac{x(m,t) - x\left(m',t\right)}{\|x(m,t) - x\left(m',t\right)\|} \quad (1)$$

as a multivariate version of the sign function. If all the machines are working properly, we expect this value to be small. Therefore, the sum of several samples over time is also expected not to grow far from zero.

Algorithm 1: The sign test.

foreach *machine* m **do**
  $S\left(m, x(t)\right) \leftarrow \frac{1}{M-1} \sum_{m' \neq m} \frac{x(m,t) - x\left(m',t\right)}{\|x(m,t) - x(m',t)\|}$;
  $v_m \leftarrow \frac{1}{T} \sum_t S\left(m, x(t)\right)$;
end
$\hat{v} \leftarrow \frac{1}{M} \sum_m \|v_m\|$;
foreach *machine* m **do**
  $\gamma \leftarrow \max\left(0, \|v_m\| - \hat{v}\right)$;
  $p(m) \leftarrow (M+1) \exp\left(-\frac{TM\gamma^2}{2\left(\sqrt{M}+2\right)^2}\right)$;
  if $p(m) \leq \alpha$ **then**
    Report machine m as suspicious;
  end
end

If all machines are working properly, the norm of $v_m = \frac{1}{T} \sum_t S(m, x(t))$ should not be much larger than its empirical mean. The p-value $p(m)$ in Algorithm 1 controls this statistic by guaranteeing a small number of false detections, depending on the significance level α.

3. ONLINE DETECTOR WITH REDUCED COMMUNICATION

We describe an online, communication-efficient version of the latent fault detector summarized in Section 2.

Detecting latent faults requires that each node must send all performance counters measured at each time point: T samples of C counters for each of the M machines. Beyond bandwidth costs, processing so much data is difficult to do on a single machine in a timely manner, due to the size and high dimensionality of the data. We apply two techniques to alleviate this issue.

Sketching is used to reduce the amount of data sent from each machine and processed by the coordinator. Instead of sending all counters, each node calculates a sketch of the said counters and sends only that. The coordinator (or monitoring node) can then perform latent fault detection using the sketches, rather than the original data. In addition to reducing the communication load, this has the added benefit of reducing the computational load, since the dimensionality of the data is greatly reduced.

The framework in Section 2.1 requires that counter values be normalized during preprocessing (step 1), and this is true as well for the sketched version[3]. We use the safe zone approach [17] to monitor both the global mean and the global variance of each counter so that they do not deviate too much from their last known values. Each machine monitors whether its data satisfies a local constraint. If all local constraints at all machines are satisfied, the global mean and variance are known not to have deviated too far from their last known values. These last known values are then used to normalize the counter values at each node, before computing the sketch. If there is any violation, the coordinator polls each node for the current mean and variance, and distributes the new global mean and variance to all nodes.

The general pseudocode is shown in Algorithm 2 and explained in detail below.

3.1 Sketches

Sketching [18, 8] is a common technique used to process large, unpredictable data streams without having to send, store and process all data. It reduces the size of the data, while still enabling queries. See [7] for a recent survey of sketched-based (and other) distributed monitoring.

For our purposes, a *sketch* is a summary function that takes a vector and transforms it to a smaller vector while approximately preserving some desired property, for example inner products [1]. We use sketches to modify our tests to greatly reduce the amount of data that must be sent and processed. For example, 200 counters could be reduced to 10 dimensions, achieving an immediate 95% reduction in size.

Formally, rather than apply test S to the set of all local counter vectors $x(m,t)$, each machine m will first apply a sketching function f to its vectors, and send only the sketch $\hat{x} = f(x(m,t))$ for processing. The modified test $\hat{S}$ will be applied to the sketches rather than the original vector: $v_m = \frac{1}{T} \sum_t \hat{S}(m, \hat{x}(t))$.

One well-suited sketch is the AMS sketch [1], which involves a random linear projection to k dimensions. In our setting, each machine would project its counter vectors to k dimensions using a specially constructed projection matrix: $\hat{x}(m,t) = f(x(m,t)) = Rx(m,t)$ where R is a random $C \times k$ matrix constructed as described in [1].

The AMS sketch is general enough so that the same sketch can be used as input to different tests. Because the sign test relies on normalized directions, and since AMS sketches are linear projections, the sign test can be applied directly to the sketch. In other words, the sum of projected vectors is

[3]Automatic counter selection (part step 1) can be done in advance, offline, using the method described in [9].

Algorithm 2: Online detection pseudocode.

OFFLINE:
Automatically select counters.

INIT / COORDINATOR SYNC:
foreach *counter i in counters* **do**
  Poll all nodes for mean and variance of counter i.
  Distribute new global mean, variance, safe zones.
end

NODE at time point t:
foreach *counter i in counters* **do**
  if *counter not in safe zone* **then**
    Violation: send local mean, variance to coordinator.
    Wait for new global mean and variance.
  end
  Let x_i = value of counter i at time t .
  Normalize x_i with last known global mean and variance.
end
Let x = vector of normalized counter values.
Compute sketch of x and send to coordinator.

COORDINATOR at time point t:
if *violation for counter i* **then**
  Run SYNC.
end
Receive sketches from all nodes.
Compute test function S on received sketches.
Add most recent test function result to v_m.
Subtract least recent test function result from v_m.
Calculate p-value for all machines and issue warnings.

the same as projecting the sum of the vectors. The resulting vector is still small for healthy machines and large for outliers. The Tukey test described in our previous work already relies on a very similar technique, and has been shown to be very effective. The LOF test depends on the distance of pairs of points. In this case, the Johnson-Lindenstrauss lemma [13] guarantees that the projection to $k = O\left(\frac{\log M}{\epsilon^2}\right)$ preserves the distances within a factor of $1 \pm \epsilon$. Since our method averages T comparisons per day in the integration phase, we can further expect that in practice the error will be smaller.

3.1.1 Sign Test on Linear Sketches

The sign test function (1) from Section 2.2 depends only on the normalized direction from $x(m,t)$ to the other vectors. Let B be the unit sphere in C dimensions. Given the assumptions in Section 2.1, for healthy machines the normalized directions to other machines tend to be distributed spherically symmetric over B, resulting in the vector $v_m = \frac{1}{T}\sum_t S(m, x(t))$ being relatively short. Conversely, for machines with consistently anomalous behavior, v_m is a relatively long vector.

Given the sketched vectors $\hat{x}(m,t) = Rx(m,t)$, the sign test is still the sum of normalized directions from $x(m,t)$, after some transformation R. We now show that applying R to the unit sphere B maintains this symmetrical distribution. Let $R = UDV^T$ be the *singular value decomposition* of R.

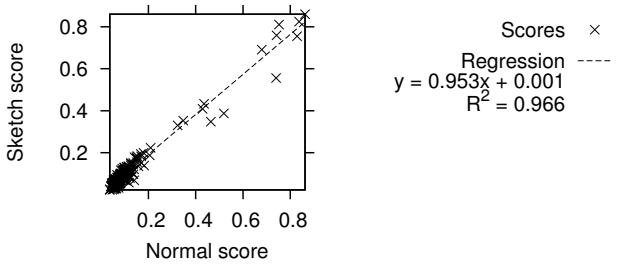


Figure 1: Sign test scores with AMS sketch compared to original scores. Sketch size is 8% of original data.

U and V^T are unitary matrices, and D is a diagonal matrix with positive elements. In geometrical terms, the transformation $R = UDV^T$ is a composition of rotation, followed by non-uniform scaling and dimensional reduction, and finally another rotation – all of which preserve the symmetric distribution around the origin. Therefore the transformation R maps the unit sphere B (in C dimensions) to an ellipsoid B' in k dimensions while preserving the symmetric distribution around the origin.

In summary, since the sign-test uses normalized directions and R preserves their symmetry around $x(m,t)$, we can apply the sign test directly to the sketched vectors $\hat{x}(m,t)$. Moreover, the sign test p-value does not depend on the dimensionality of the vectors, and so we can use it as is.

Preliminary experiments on counter logs from a small sample of 260 machines in a single day show that sign test scores and p-values computed on sketched data match the original very well. Figure 1 shows a comparison of sign test scores based on AMS sketches to regular (centralized, or parallel) sign test scores. The figure and linear regression show that the scores match very well, with $R^2 = 0.966$, very close to 1. The sketch reduced the data size by 92% – from 123 counters to 10 dimensions. The p-values are similarly close to the original values.

3.1.2 Online Integration Using a Sliding Window

The integration phase in stage 2 of the framework in Section 2.1 computes $v_m = \frac{1}{T}\sum_t S(m, x(t))$. Computing $S(m, x(t))$ only requires the data from time t, and therefore it is trivial to turn any test into an online test by keeping a window of test function (S) outputs for the last T sketches sent from the monitored machines. When new data arrives at time t, the coordinator updates the current v_m by computing and adding $\frac{1}{T}S(m, \hat{x}(t))$, and subtracting the least recent stored test result, $\frac{1}{T}S(m, \hat{x}(t-T-1))$. The p-value for each machine in the time window can then be computed in the usual manner. Since the test function S need only be computed for the most recent time, and since the sketches are of low dimension k, processing and memory costs are low. This allows the computation to be done on a single coordinator machine on time, before the next round starts.

3.2 Scaling By Monitoring Variance

Our tests require the data to be standardized during preprocessing: each counter should be globally centered to zero mean and unit variance. In some settings we can assume that a counter's mean and variance do not change much,

or that they have a daily cycle. However, we might wish to avoid that assumption, and handle unpredictable workloads.

We use the *safe zones* approach [17, 21] to monitor both the global mean and the global variance of each counter. In this approach, each monitored machine receives a local constraint on its data $x(m,t)$ from a coordinator machine, such that if all local constraints are satisfied, the global monitored value $f(x(t))$ for some function f of the global aggregate is within a pre-defined threshold. Violations of local constraints are sent to the coordinator machine, which resolves them and sends updated local constraints to participating machines.

Given the last known global mean and variance of the last T samples, we define some lower and upper threshold, for example 0.9 and 1.1 times the last known values. If there is any violation, the coordinator polls each node for the current mean and variance, and distributes the new global mean and variance to all nodes. We can trade-off accuracy and communication by adjusting the high and low thresholds when monitoring. Violations are less likely if global mean and variance are allowed to drift further from their last known values – reducing communication but also decreasing accuracy [17].

We monitor each counter independently, so it is enough to show how we monitor a single counter X. Further note that all tests described in [9] are invariant to data translation, and so we do not monitor the global mean explicitly.

3.2.1 Notations

The set of values of counter X over the last T times and over M nodes (machines) is denoted by $X(t)$. We denote by $X_i(t)$ the values of X at node i for the last T times up to t. Thus $\mathrm{E}[X_i(t)]$ is the mean of the last T values at node i in time t, while $\mathrm{E}[X(t)]$ is the global mean of the last values at all nodes. Denote $\mu_i(t) = \mathrm{E}[X_i(t)]$ the local means, and $\mu(t) = \mathrm{E}[X(t)]$ the global mean. Similarly, we denote $\lambda_i = \mathrm{E}\left[X_i(t)^2\right]$, the local mean of the squares, and $\lambda = \mathrm{E}\left[X(t)^2\right]$ the global mean. Let $V(t) = (\mu(t), \lambda(t))$, and $V_i = (\mu_i(t), \lambda_i(t))$, the global and local monitored vectors, respectively.

3.2.2 Monitoring

We wish to monitor the global variance $\mathrm{Var}(X)$ at each time t. Recall that:

$$\mathrm{Var}(X) = \mathrm{E}\left[X^2\right] - (\mathrm{E}\left[X\right])^2 = \lambda - \mu^2 \quad .$$

We therefore monitor the conditions $L \leq \lambda - \mu^2 \leq H$, for some lower and upper variance thresholds L and H. Figure 2 shows the *admissible region* (the region in which the conditions hold), $0.5 \leq \lambda - \mu^2 \leq 1.5$. Following [17], we aim to find a convex safe zone G which is contained within the admissible region. Since convex sets are closed under averaging, when all local vectors are inside the safe zone, the global mean is guaranteed to be inside as well.

Let $t = 0$ be the last global synchronization time, and let $V(0) = (\mu(0), \lambda(0))$ be the *reference point*, the last known global mean and mean-of-squares, computed that time. For each node i we define the local drift vector $d_i(t)$ as the drift of the current vector from the node's vector during the last synchronization: $d_i(t) = V_i(t) - V_i(0)$.

Since we wish to monitor that the global $V(t)$ is within some convex set G, we define equivalent local conditions on the drift vectors. The current local vectors can be written

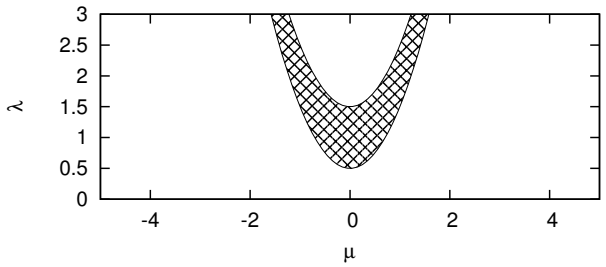


Figure 2: Admissible region for $L = 0.5, H = 1.5$.

in terms of drift vector d_i: $V_i(t) = V_i(0) + d_i(t)$. Note that the global vector is the mean of the local vectors, and can therefore be written as the mean of drifts and the reference point:

$$V(t) = \frac{1}{M}\sum_i V_i(t) = V(0) + \frac{1}{M}\sum_i d_i(t) \quad . \qquad (2)$$

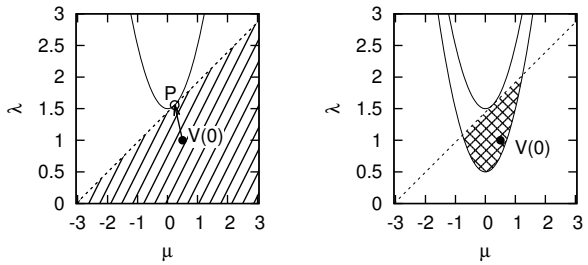
Let $W_i(t) = V(0) + d_i(t)$ be the local drift from the last reference point. Note that $V(t) = \frac{1}{M}\sum_i W_i$, recall G is convex, and from (2) we arrive at the local conditions: if $\forall i, W_i \in G$ then $V(t) \in G$.

To monitor that the variance is between L and H, we derive separate safe zones: one for variance above L and another for variance below H. As long as the local conditions for both safe zones are maintained in all nodes, we are guaranteed that the variance is within the allowed range.

Variance Above Lower Threshold. We wish to define a convex safe zone G_L so that as long as $V(t) \in G_L$ then $\mathrm{Var}(X) \geq L$. This corresponds to monitoring that $\lambda - \mu^2 \geq L$, which is already a convex set – the area above a parabola – and can be directly used as safe zone. Therefore the local condition for each node i is trivial: $I_i(t) \in G_L$: $I_i(t) = V(0) + d_i(t) = (a, b)$ and monitor that $b - a^2 \geq L$.

Variance Below Upper Threshold. We wish to define a convex safe zone G so that as long as $V(t) \in G$ then $\mathrm{Var}(X) \leq H$. This area is the area below a parabola, which is not a convex set. However, we can find a tangent half-plane I below this parabola. This half-plane is a convex set, and since $I \subset G$, then as long as $V(t) \in I$, $V(t) \in G$ and therefore $\mathrm{Var}(X) \leq H$.

We use the reference point $V(0)$ to find the optimal hyperplane. The thresholds H and L are reset during synchronization, so obviously $V(0) \in G$. We can choose any half-space I such that $V(0) \in I$, but to avoid future unnecessary synchronization we choose I such that $V(0)$ is far from the boundary of G. Doing so ensures that drift has to be large to cause a violation. Consequently, we choose I as the tangent at point P, where P is the closest point to $V(0)$ on the parabola $\lambda - \mu^2 = H$, and the local condition is $W_i \in I$. We can find P numerically, or by minimizing the distance from the parabola to $V(0)$. For example, if $V(0) = (0.5, 1)$ and $H = 1.5$, then the closest point on the parabola is $\mu \approx 0.237$. This yields the point $P = (0.237, 1.556)$, and finally the induced safe zone I: the half-plane $\lambda - 0.474\mu < 1.443$. Figure 3(a) shows $V(0)$, P and the resulting safe zone, and Figure 3(b) shows the intersection with the safe zone for the lower limit $L = 0.5$.



(a) Upper threshold. **(b) Upper and lower.**

Figure 3: Safe zones for $L = 0.5, H = 1.5$ where $V(0) = (0.5, 1)$.

3.2.3 Handling Violations

If one of the local conditions $W_j \in G$ is violated, it may be because $\mathrm{Var}(X)$ is no longer in the range, or due to a false alarm. The simplest way to deal with a violation is to perform a global synchronization: each node sends its current $V_i(t)$ to the coordinator. The coordinator "resets the time" to $t = 0$, computes the new global reference point $V(0)$, and sends it to the nodes, where it is used for monitoring and scaling.

In terms of communication, our synchronizations are fairly inexpensive. Each node sends only two numbers per counter (μ and λ), rather than the entire time window of T samples. They also improve the accuracy of scaling, since nodes have fresh global mean and variance. There are safe zone techniques that allow partial synchronization for further communication reduction, for example by balancing a node with local violation with another node that has enough slack [2].

4. FUTURE WORK

This work uses sketching and safe zones to adapt the latent fault detector in [9] to a streaming setting, resulting in an online, communication-efficient outlier detector for common scale-out systems. Preliminary results show that the adapted detector obtains very similar results to those of the original latent fault detector for the sign test. Future work will concentrate on adapting additional tests, evaluating the detector on real-world systems, and exploring the communication-accuracy trade-off.

5. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Seventh Framework Programme under grant agreement Nº 255951.

6. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 1999.

[2] D. Ben-David. Violation resolution in distributed stream networks. Master's thesis, Technion I.I.T, 2012.

[3] P. Bodík, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proc. EuroSys*, 2010.

[4] G. Bronevetsky, I. Laguna, B. De Supinski, and S. Bagchi. Automatic fault characterization via abnormality-enhanced classification. In *Proc. DSN*, 2012.

[5] H. Chen, G. Jiang, and K. Yoshihira. Failure detection in large-scale internet services by principal subspace mapping. *IEEE Trans. Knowl. Data Eng.*, 2007.

[6] I. Cohen, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. OSDI*, 2004.

[7] G. Cormode. The continuous distributed monitoring model. *SIGMOD Rec.*, 2013.

[8] G. Cormode and M. Garofalakis. Sketching probabilistic data streams. In *SIGMOD*, 2007.

[9] M. Gabel, A. Schuster, R.-G. Bachrach, and N. Bjorner. Latent fault detection in large scale services. In *Proc. DSN*, 2012.

[10] C. Huang, I. Cohen, J. Symons, and T. Abdelzaher. Achieving scalable automated diagnosis of distributed systems performance problems. Technical report, HP Labs, 2007.

[11] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 2007.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, 2007.

[13] W. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in modern analysis and probability (New Haven, Conn., 1982)*, Contemporary Mathematics. 1984.

[14] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proc. FAST*, 2010.

[15] S. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *Proc. DSN*, 2012.

[16] S. Kavulya, R. Gandhi, and P. Narasimhan. Gumshoe: Diagnosing performance problems in replicated file-systems. In *Proc. SRDS*, 2008.

[17] D. Keren, I. Sharfman, A. Schuster, and A. Livne. Shape sensitive geometric monitoring. *Knowledge and Data Engineering, IEEE Transactions on*, 2012.

[18] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 2005.

[19] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *Proc. ICDM*, 2008.

[20] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan. Vigilant: out-of-band detection of failures in virtual machines. *SIGOPS Oper. Syst. Rev.*, 2008.

[21] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. *TODS*, 2007.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. SOSP*, 2009.

[23] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proc. DSN*, 2005.

Elastic Complex Event Processing under Varying Query Load

Thomas Heinze[1] Yuanzhen Ji[1] Yinying Pan[1] Franz Josef Grueneberger[1]
Zbigniew Jerzak[1] Christof Fetzer[2]

[1]SAP AG
Dresden, Germany
firstname.lastname@sap.com

[2]System Engineering Group, TU Dresden
Dresden, Germany
christof.fetzer@tu-dresden.de

ABSTRACT

Distributed data stream processing systems, like Twitter Storm or Yahoo! S4, have been primarily focusing on adapting to varying event rates. However, as these systems are becoming increasingly multi-tenant, adaptation to the varying query load is becoming an equally important problem.

In this paper we present FUGU – an elastic allocator for Complex Event Processing systems. FUGU uses bin packing to allocate continuous queries to a varying set of nodes. Driven by elasticity requirements FUGU maximizes the overall system utilization while trying to maintain stable processing latencies.

The specific contributions of this paper are: (1) introduction of a re-balancing scheme for bin packing allowing FUGU to increase overall system utilization by six percent and (2) a detailed study of achievable system utilization and latency under real-life workload from Frankfurt Stock Exchange.

1. INTRODUCTION

Distributed complex event processing (CEP) has been commonly used in context of financial trading systems [1]. Typical CEP use cases in financial domain usually revolved around single user, single query usage pattern. However, with recent proliferation of CEP in industries such as manufacturing [8] or analytics [11] the usage pattern is switching towards multiple users, multiple queries per system. The implication of this trend is the need for CEP systems to be able to accommodate not only varying event load but also varying query load.

In order to avoid constant overprovisioning and to be able to handle sudden load surges distributed CEP systems must be able to scale both in and out. Being able to scale both in and out while maintaining high overall system utilization is the ultimate goal of an elastic system [2]. Elasticity is an important property of every distributed system as it ensures its economic feasibility while being executed on any cloud platform.

Several authors have studied building elastically scalable complex event processing systems [7, 12]. However, we are not aware of a work which would explicitly target the problem of the varying query load in elastic CEP systems. In this paper we present the design and evaluation of the elastic allocation component – FUGU. FUGU can dynamically allocate and de-allocate both stateless and stateful queries in order to meet the utilization goals. To that end FUGU relies on bin packing to allocate queries to hosts.

The contributions of this paper are following: (1) we present a re-balancing extension of a state of the art bin packing approach [4], which allows to improve the average utilization of the system by up to 6% and (2) we present a detailed evaluation of the achievable utilization as a function of a given utilization target. The evaluation of our elastic allocation component has been performed on top of a commercial distributed complex event processing system using tick data streams from Frankfurt Stock Exchange.

2. SYSTEM ARCHITECTURE

Figure 1 shows the FUGU component and its interaction with the underlying CEP system. The underlying CEP system consists of several instances of a CEP engine running in parallel on heterogeneous hosts. The CEP system accepts and processes continuous queries consisting of direct acyclic graphs of operators. Our system supports primitive relational algebra operators (selection, projection, join, aggregation) as well as additional CEP specific operators (sequence, source and sink). Each operator can be executed on an arbitrary host. Therefore, the computation of a query can be partitioned over multiple hosts. The number of hosts is variable and dynamically adapted to the changing resource requirements by the FUGU component. FUGU is always provisioning one or two hot hosts to allow for a fast scale out [6].

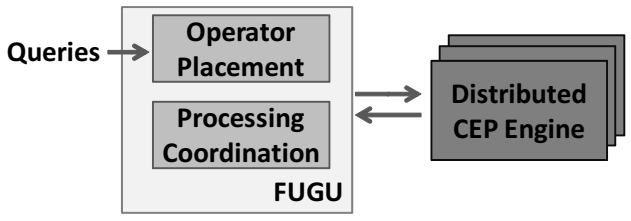


Figure 1: System architecture

FUGU is a centralized component. The role of FUGU is twofold: (1) it coordinates different instances of the CEP

engine and (2) it calculates placement decisions. When a new query is added or an existing query is removed, a bin packing algorithm is used (see Section 3) to calculate the operator to host assignment. When a new operator needs to be placed, FUGU will always try to locate a host with enough available resources to host this operator. If no such host can be found, a new host will be assigned to the system. When all operators on a certain host are removed, the host is then released by the system.

As soon as such an assignment has been derived, FUGU coordinates the placement of new and re-placement of existing operators. To that end FUGU communicates with all involved hosts using a topic-based publish/subscribe protocol. Newly added operators subscribe to their predecessor operators. Data published by an operator is sent to all subscribers. FUGU supports re-placement of both stateless (source, filters, projection, sinks) as well as stateful operators (aggregation, join, sequence) using a state transfer protocol similar to the one of [13].

3. OPERATOR PLACEMENT

The foundation of our operator placement approach is a load model, which estimates and measures CPU, memory and network consumption for each individual operator. When new queries are added, all variables in the model are first estimated using a worst case assumption. These values are subsequently updated during runtime with precise measurements.

The required CPU load ($load_{\text{CPU}}$) for a given operator (op) is calculated based on the operator's input rate ($input(op)$) and its per event processing time ($proc(op)$):

$$load_{\text{CPU}}(op) = proc(op) \cdot input(op) \tag{1}$$

During the estimation phase, we assume that the processing time of a new operator is comparable to the processing time of currently running/previously executed operators of the same type. The input rate is derived based on the input rate of the predecessor operators and estimations of their selectivities in a fashion similar to the approach presented by Viglas et al. [14]. For the purpose of the estimation we constantly measure the source input rate and use the maximum value observed so far. The major advantage of this scheme is that it only requires the input rates of the sources.

We use similar approach to estimate operators' memory and network consumption. The network bandwidth is derived from the operators' input and output rates, their selectivity (predicate) and the average size of input and output events. The memory consumption is estimated using a linear model which multiplies the operators' event rate by the window size and event size. The network consumption model is placement-aware: operators placed on the same host are assumed to communicate via in memory message passing. Operators on different hosts are assumed to communicate via network.

3.1 Elastic Operator Placement

The placement is calculated using a global bin packing algorithm [5] in a fashion similar to the one proposed by Backman et al. [4]. Bin packing algorithm calculates an assignment of items (operators) to bins (hosts) in a way that a minimal number of bins is used. The major criteria for assignment is the required CPU load of an operator. In addition, hosts with insufficient memory or network bandwidth are removed from the list of potential target hosts. FUGU uses a FirstFit bin packing variant, which assigns a newly added operator to the first node with enough remaining capacity.

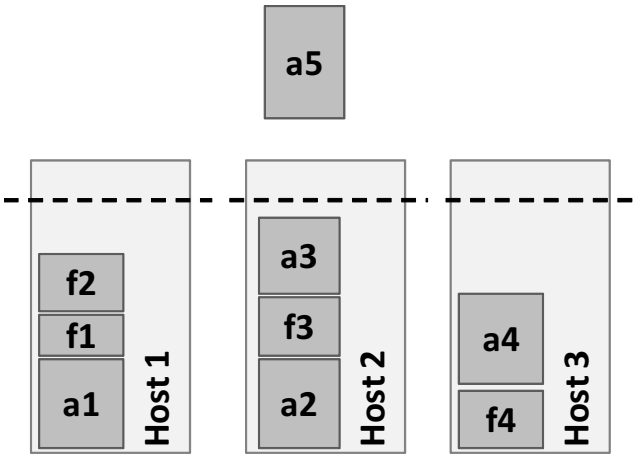


Figure 2: Example for applying the rebalancing heuristics

The placement algorithm can be configured to aim for a certain target utilization. This is realized by an additional user-defined parameter: the utilization threshold $thres$. The $thres$ value is used as the available capacity of a host, which should not be exceeded by the bin packing algorithm.

3.2 Re-balancing Heuristics

The above bin packing approach allows to scale out and to scale in with a changing number of queries. However, after evaluating this approach we have observed that the system is often reporting suboptimal utilization values – see Section 4. This is caused by the fact that remaining operators are scattered across all hosts in the system. This, in turn, prevents FUGU from releasing these hosts.

There exist two alternative approaches towards solving this issue. Either the bin packing algorithm is re-executed for all operators left in the system or specific operators are selected and re-placed so as to release least loaded hosts. A re-execution of the bin packing approach with all remaining operators would provide the best solution, however, it would also result in a large amount of operators and state being moved. This in turn would negatively impact the availability of the system. Therefore, in order to minimize the impact on the system availability we have implemented a re-balancing approach.

As soon as a query is removed, the re-balancing algorithm calculates the currently required minimal number of hosts ($host_{\min}$):

$$host_{\min} = \left\lceil \frac{\sum_{\forall op} load_{\text{CPU}}(op)}{thres} \right\rceil \tag{2}$$

In case the current number of hosts used by the system is larger than the calculated minimal number of hosts ($host_{\min}$) a re-balancing is triggered. During re-balancing only operators from hosts with the minimal load are subject to bin packing. Bin packing is executed for these operators until the total number of used hosts reaches $host_{\min}$.

An additional heuristic is used to detect imbalance during addition of queries. Let us consider the scenario shown in

Figure 2, where three active hosts are used and a new operator $a5$ should be placed. None of the hosts has enough remaining capacity to allow an assignment of the operator $a5$. Therefore, a new host needs to be allocated and the $a5$ operator needs to be placed on this host. However, if we consider the total remaining capacity on all hosts it should be possible to place the operator without allocating any new hosts. The re-balancing is triggered if during the addition of an operator op a new host should be allocated and the following condition holds:

$$load_{\text{CPU}}(op) < (n \cdot thres - \sum_{\forall o} load_{\text{CPU}}(o)) \qquad (3)$$

where n is the number of currently active hosts in the system and $\forall o$ represents all operators currently running in the system.

For re-balancing we choose the host, where the difference between remaining capacity and the newly assigned operator load is minimal. For this host we use a subset algorithm [10] to identify a minimal set of operators to redistribute in order to make place for the new operator to be added. We use the algorithm to calculate all valid solutions with a summed CPU load within the interval $[load_{\text{CPU}}(op), load_{\text{CPU}}(op) + int]$, where int describes the interval size. From this set we select the solution, which requires the smallest amount of state to be moved. Considering the example in Figure 2, Host 3 will be selected as one with the closest remaining capacity. Subsequently, operator $f4$ will be selected and moved to Host 1 and operator $a5$ will be placed on Host 3.

4. EVALUATION

We have implemented FUGU on top of a state of the art, commercial, distributed CEP engine. We have extended the underlying CEP system with capabilities required for dynamic host addition and removal as well as state migration. The evaluation is conducted in a shared, private cloud environment with up to 10 hosts with 2 cores and 4 GB RAM each. For evaluation we use a real-world tick stream from the Frankfurt Stock Exchange. We can replay the tick stream with a variable or a fixed data rate. For evaluating our system we use the following query template:

```
SELECT avg(price) FROM tickStream WITHIN x SEC
GROUP BY comp WHERE sector=y;
```

The above query calculates the average price for each company within a certain sector. The query workload is made variable by choosing the window size (x) and the sector (y) randomly. The query workload pattern was extracted from a web server log [3] – see Figure 3(a) and 3(b).

Performance is evaluated based on the end to end latency. We define the end to end latency as the difference between the time an event enters the system via source operator and the time it leaves the system via sink operator. Due to different complexities of queries the end to end latency of different queries can not be easily compared. Instead, for each query we calculate the ratio between the initial latency measured for the first ten seconds after the query has been added and the current end to end latency. We label this value as *latency ratio*. Latency ration should be ideally always equal to 1.

4.1 Elastic Scaling of FUGU

The goal of the first experiment is to demonstrate that FUGU is able to elastically scale the underlying CEP system

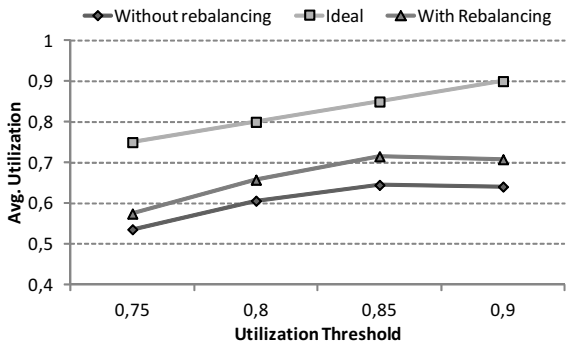


Figure 4: System utilization as a function of utilization threshold *thres*

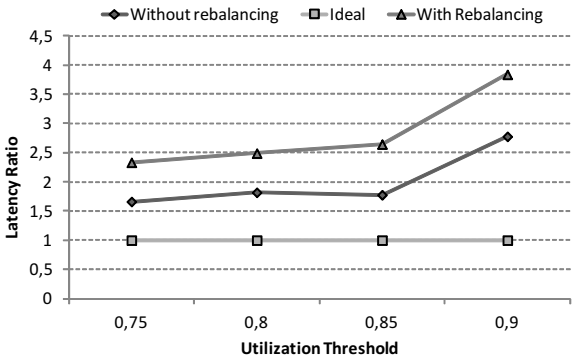


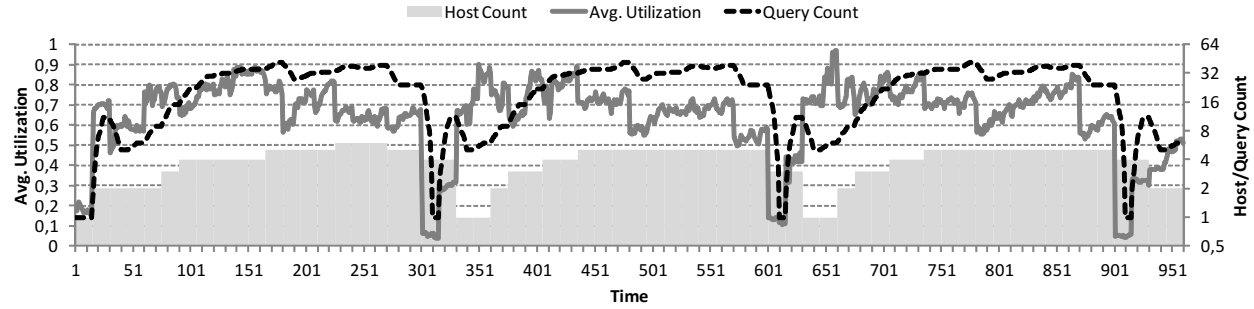
Figure 5: Latency ratio as a function of utilization threshold *thres*

with a varying number of queries. For this experiment we set the utilization threshold *thres* to 0.85. Figure 3(a) shows the average system utilization and used hosts count as a function of the query count. During peak load system runs 45 queries in parallel across six hosts. It can be observed that FUGU automatically scales underlying CEP system out and in depending on the query workload. The average utilization remains constant and oscillates around 60%. Figure 3(c) shows the corresponding maximum latency ratio across $a$ll queries running in the system. The average latency ratio of all queries stays close to 1, however certain queries experience short latency peaks. According to expectations this behavior manifests itself mainly during reconfigurations of the system, i.e., addition or removal of hosts.

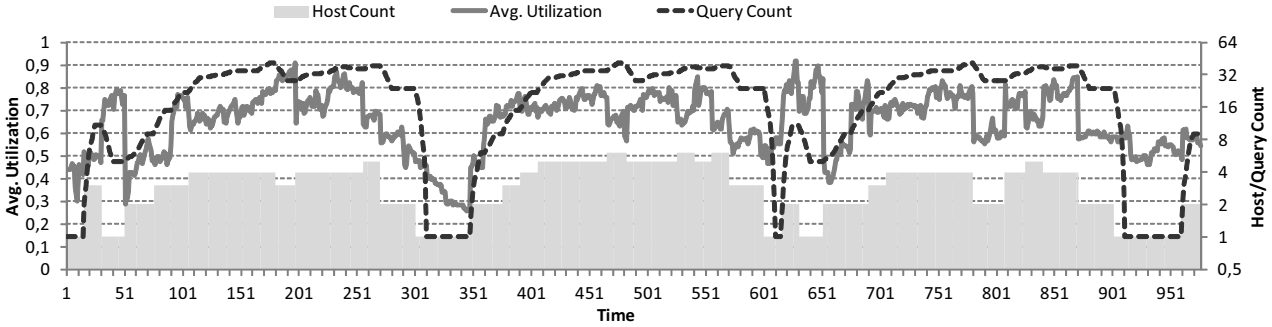
In the following experiment we have enabled the re-balancing algorithm and re-executed the experiment. Figure 3(b) shows that the system is able to release hosts earlier and in average uses less hosts than the approach without re-balancing. The average utilization increases to 65%. However, due to the re-balancing more peaks in the latency ratio can be observed – see Figure 3(d). This confirms the existence of a basic intuitive trade-off: the more aggressive the elasticity policy the less stable the system becomes.

4.2 Achievable Utilization

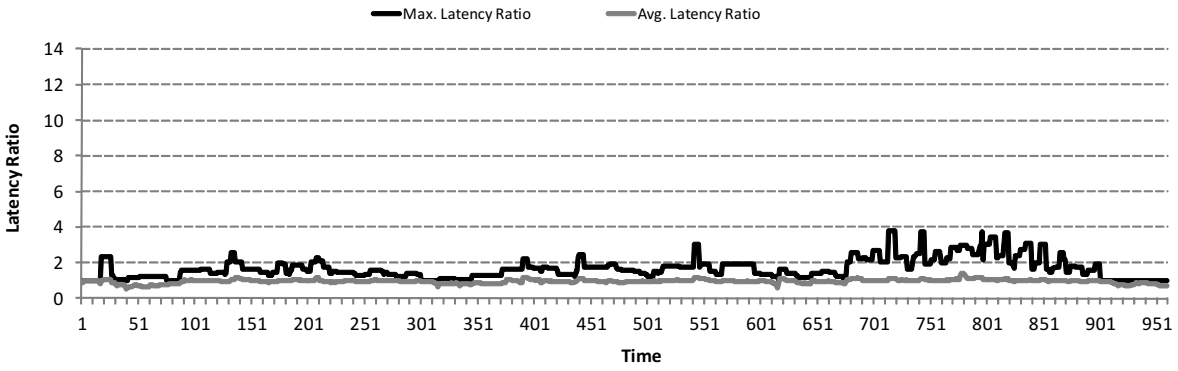
The goal of FUGU is to maximize the system utilization without significantly impacting the end to end latency of the running queries. In order to study the maximal achievable utilization of our system we changed the threshold *thres*



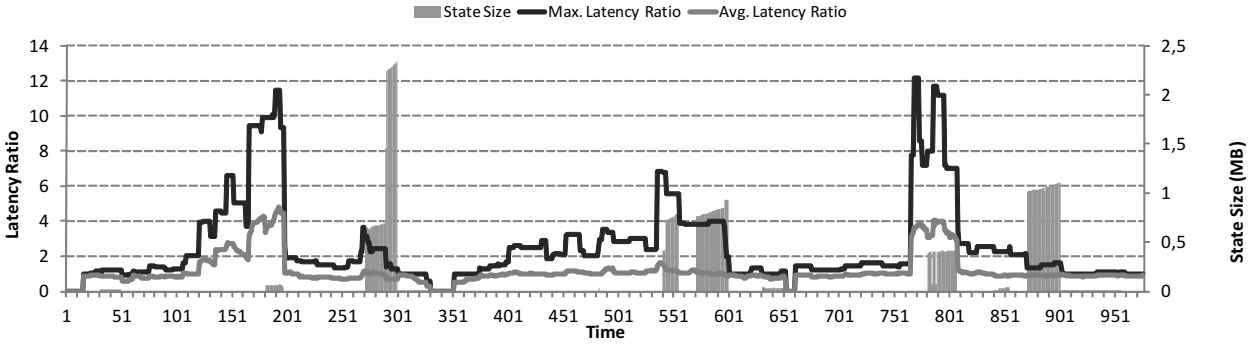
(a) Elastic scaling – average system utilization as a function of the query count



(b) Elastic scaling with re-balancing – average system utilization as a function of the query count



(c) Elastic scaling – maximum and average latency ratios



(d) Elastic scaling with re-balancing – maximum and average latency ratios and migrated state size

Figure 3: Elastic scaling with and without re-balancing

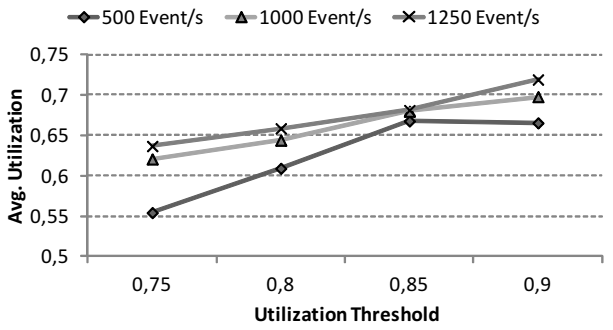


Figure 6: Average utilization as a function of varying event rate and utilization threshold

for the upper bound of utilization per host from 75% up to 90%. Figure 4 shows the resulting system utilization as a function of the threshold *thres*. In addition, Figure 5 shows the average latency ratio. We can observe that the achievable utilization increases from 53% for $thres = 0.75$ to 64% for $thres = 0.9$ while the latency ratio increases from 1.6 to 2.7. The maximal achievable utilization saturates starting from a value of utilization threshold $thres = 0.85$.

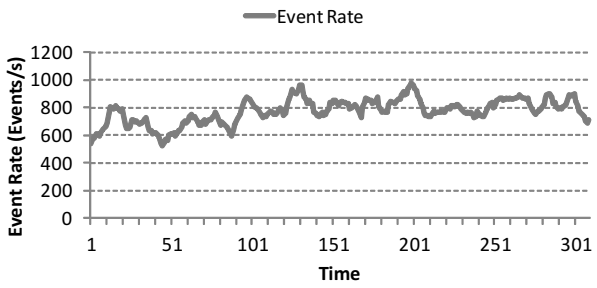
By using the re-balancing scheme the average utilization can be improved by up to six percent points, e.g. for $thres = 0.9$ to 70%. The maximal latency ratio increases to 3.8. The maximal latency ratio is proportional to the frequency with which the re-balancing is executed.

4.3 Influence of Event Rate

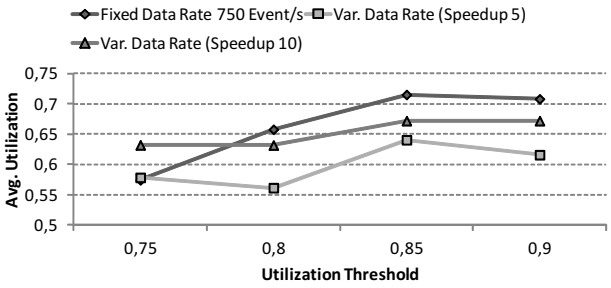
We have also measured the achievable utilization as a function of different stable event rates – see Figure 6. We have varied the event rate between 500 and 1250 events per second. The number of queries is identical as in case of the previous experiment. The number of hosts is automatically changing from 3 hosts for 500 events per second run with threshold 0.9 up to 9 hosts for 1250 events per second run with threshold of 0.75. From the experiment we can conclude that no linear correlation between the input rate and the achievable utilization can be drawn. This indicates that setting a good utilization threshold for different system conditions is a challenging problem.

To emphasize this result we re-ran above experiment with a varying event rate – see Figure 7. For this experiment we have fixed the utilization threshold at 0.9. The event rate pattern over time is shown in Figure 7(a). The event rate changes between 300 and 600 events per second for a speedup value of 5, and between 400 and 1000 events per second for a speedup value of 10.

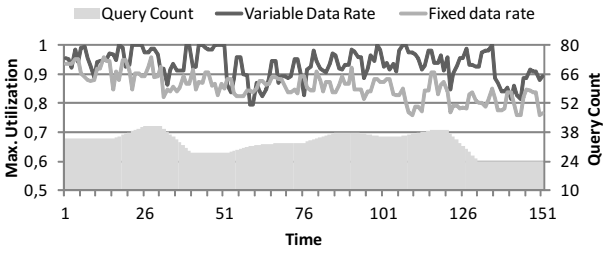
Figure 7(b) shows that the average system utilization, in case of variable event rate, is lower than in case of a fixed event rate. It is also, to a large extent, independent of the selected utilization threshold. Moreover, we have observed (see Figure 7(c)) that for individual hosts the utilization threshold is often exceeded. These two observations show a need to combine our approach with run-time adaptation and elasticity policies [6, 7], in order to be able to efficiently handle varying event rates.



(a) Variable event rate pattern for a speedup value of 10



(b) Average system utilization as a function of the utilization threshold and data rate



(c) Maximum utilization of an individual host

Figure 7: Variable event rate evaluation

4.4 Discussion

Based on the above evaluation we can conclude that our approach is well suited for elastic scaling with a varying number of queries. The system under control of FUGU is able to dynamically adjust the number of hosts and is able to keep the latency ratio close to 1 for the presented scenario. By trying to maximize the utilization we have also demonstrated that a trade off between latency ratio and achievable utilization exists. Specifically, finding a good upper threshold for the utilization of the system seems to be both important and non-trivial.

We have also outlined, that the event rate has a major influence on the achievable utilization. Especially, in case of varying event rates the system utilization significantly decreases. This requires the addition of run-time adaptation to FUGU, which we consider as future work.

5. RELATED WORK

Elasticity in context of data stream processing systems has been studied by various authors [6, 7, 12], however, none

of the proposed approaches considered a varying query load. Schneider et al. [12] present a scheme for elastic resource scaling within a single node. The system can adapt the number of threads used by a single operator to be able to handle varying event rate. Other approaches focus on adapting a distributed data stream processing system to changing event rates. Gulisano et al. [7] describe a distributed system using an upper and a lower bound on the load variance to trigger operator migration whenever these bounds are violated. The implication of this approach is the possibility of allocation of new hosts and thus worsening of the overall system utilization. Fernandez [6] et. al. present an integrated solution for dynamic scale-out and fault tolerance. Presented system supports check-pointing-based fault tolerance and policy-based scale out. However, it is not possible to scale the system in, therefore, unlike FUGU, it cannot be considered as fully elastic.

Balancing the load among hosts of a streaming system is related to a class of algorithms used for operator placement [15, 4]. Operator placement algorithms can target different objectives, most common being: end to end latency, network bandwidth and load (im-)balance – see [9] for a comprehensive survey of placement strategies. Xing et al. [15] presents an algorithm which balances the load between all hosts of the system by minimizing the load variation between hosts. FUGU uses similar technique where an initial assignment is optimized by partial re-balancing. However, the approach of Xing et al. only works for a fixed number of hosts, whereas FUGU can adjust the number of hosts dynamically. Backman et al. [4] present an approach, which balances the load between hosts using bin packing. Using simulation Backman et al. conclude that the system is able to provide latency guarantees. Evaluation with FUGU demonstrates that this claim is difficult to uphold in a system with a dynamic set of queries. Moreover, it is in opposition to the high utilization goal of elastic systems.

6. CONCLUSION

In this paper we have presented FUGU, an allocation component for distributed complex event processing systems. FUGU is able to elastically scale in and out the underlying CEP system with a varying query load. We have evaluated FUGU using real life workloads and demonstrated that it can achieve a good average utilization with a stable latency ratio. We have also presented a re-balancing extension allowing to migrate stateful and stateless operators between hosts, thus improving the overall system utilization by up to 6%.

For the future we plan to investigate how to improve the ratio between achievable utilization and measured latency. We also plan for provisioning QoS guarantees for a system under the control of FUGU. In addition, we want to extend the system to allow for run-time adaptation to dynamically changing event rates.

7. REFERENCES

[1] A. Adi, D. Botzer, G. Nechushtai, and G. Sharon. Complex event processing for financial services. In *SCW 2006: Proceedings of the 2006 IEEE Services Computing Workshops*, pages 7–12, 2006.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[3] M. Arrington. AOL proudly releases massive amounts of private data. *TechCrunch: http://www. techcrunch. com/2006/08/06/aol-proudly-releasesmassive-amounts-of-user-search-data*, 2006.

[4] N. Backman, R. Fonseca, and U. Çetintemel. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, page 1. ACM, 2012.

[5] E. Coffman Jr, M. Garey, and D. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[6] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, 2013.

[7] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[8] Y. Ji, T. Heinze, and Z. Jerzak. HUGO: Real-Time Analysis of Component Interactions in High-Tech Manufacturing Equipment. In *DEBS 2013: Proc. Of the 7th ACM International Conference on Distributed Event-Based Systems*, 2013.

[9] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *Internet Computing, IEEE*, 12(6):50–60, 2008.

[10] S. Martello and P. Toth. Algorithms for knapsack problems. *Surveys in combinatorial optimization*, 31:213–258, 1987.

[11] C. Mutschler, H. Ziekow, and Z. Jerzak. The DEBS 2013 grand challenge. In *DEBS 2013: Proc. Of the 7th ACM International Conference on Distributed Event-Based Systems*, 2013.

[12] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[13] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36. IEEE, 2003.

[14] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 37–48. ACM, 2002.

[15] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 791–802. IEEE, 2005.

Adaptive Selective Replication for Complex Event Processing Systems

Vision Paper

Franz Josef Grüneberger
SAP AG
Chemnitzer Str. 48
01187 Dresden, Germany
franz.josef.grueneberger@sap.com

Thomas Heinze
SAP AG
Chemnitzer Str. 48
01187 Dresden, Germany
thomas.heinze@sap.com

Pascal Felber
University of Neuchâtel
Switzerland
pascal.felber@unine.ch

ABSTRACT

As of today, active replication is used in complex event processing systems to enable near zero latency take over in case of host failures. Moreover, elastic complex event processing systems adapt their resource consumption to the actual system load. However, active replication is a coarse-grained approach demanding the duplication of all used resources. Therefore, we envision a system adopting adaptive fine-grained replication strategies allowing to trade off availability and used resources.

1. INTRODUCTION

The dissemination of high frequency event sources raises the demand to extract information from high velocity data streams. Prevalent application domains comprise automatic stock trading, credit card fraud detection, automated homecare, as well as scientific experiments, logistics, and telecommunication. To process high velocity data (>10.000 events per second) with low latency ($<100\,\text{ms}$), a new class of applications enabling the efficient analysis of data in real-time has ermerged. Prominent examples of such complex event processing systems comprise Borealis [1], IBM System S [9], and Yahoo! S4 [17].

Distributed complex event processing systems spanning thousands of hosts cope with very high data rates and extensive computations. However, the error probability increases with the number of components in a system. Because in data centers the probability for a single host to fail at least once a year is between 4 and 8 percent [6, 20] and distributed complex event processing systems execute all computations in memory, fault tolerance techniques have to be leveraged to circumvent unrecoverable data loss.

Various fault tolerance techniques like active replication [2] and check pointing [11, 14] have been studied in the context of complex event processing systems. To ensure failover with almost zero latency, we focus on active replication. However, actively replicating a system requires at least twice the resources. Therefore, this paper envisions techniques and outlines the major challenges for an elastic fault-tolerant complex event processing system that achieves high availability via adaptive selective replication. Specifically:

1. We present an approach that leverages spare resources to increase the availability of queries by replicating selected parts, i. e. operators, of the running system. Specifically, we present different strategies to select operators for replication. Moreover, striving for maximal availability, we introduce different placement strategies to deploy operators on hosts.

2. We envision an optimization component supporting the replication of queries to hit a certain availability target while adhering to resource constraints. Moreover, the component should assist minimizing the resource usage for a certain availability goal.

3. We explore the challenges arising from replication in elastic distributed complex event processing systems, where both queries and hosts are dynamically added to and removed from the system.

The remainder of this paper is structured as follows: Section 2 introduces the assumed system model. Section 3 presents an approach leveraging spare resources to improve the availability of queries. In Section 4 we propose an optimization component to minimize the resource consumption for a certain availability target. The challenges arising from an application of the approaches in an elastic system are outlined in Section 5. Related research is discussed in Section 6. Finally, Section 7 concludes the paper.

2. SYSTEM MODEL

2.1 Query Model

Queries in the system are continuous queries that can be added and removed at any point in time. As opposed to one-shot queries that are sent to the system and then produce a result once, continuous queries remain in the system for a certain period of time and produce results continuously. Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_l\}$ be the set of queries in the system.

Queries are specified by the user in an event processing language (EPL). In the system queries are represented as

directed acyclic graphs (DAGs). Nodes represent operators that are connected by unidirectional edges. A query compiler transforms queries specified in EPL into a query graph.

Each operator has one or two inputs and multiple outputs. Operators with two inputs are referred to as binary operators. Each operator has a type defining its basic behaviour: source, projection, filter, aggregation, sequence, join, and destination. Besides a type all operators, except sources and sinks, have a predicate refining its functionality. For example in case of a filter operator the predicate specifies the filter condition.

The following example query calculates the average price of the SAP stock over the last 10 minutes.

```
INSERT INTO outStream
   SELECT compName, avg(tick)
   FROM tickStream WITHIN 600 seconds
   GROUP BY compName
   WHERE compName = "SAP";
```

The corresponding query graph, which is depicted in Figure 2, contains a source, filter, aggregation, and destination operator.

To minimize the number of operators in the system, all queries are optimized via a query optimization component, which analyses the query graph of already running queries with respect to reusable operators leveraging incremental multi query optimization (MQO) techniques [12]. The optimizer maintains an internal global query graph subsuming all currently deployed queries. When adding a new query, reusable parts are discovered using a breadth-first search on the global query graph. Figure 1 shows an example for multi query optimization with two queries. Operators of the already running query are depicted in the upper lane, whereas operators for the newly added query are depicted in the lower lane. Assuming that the same operator name indicates the same operator type and predicate, the operators S_1 and F_1 are part of both queries. Instead of redeploying this operators, they are reused from the already running query, which is illustrated via the grey box.

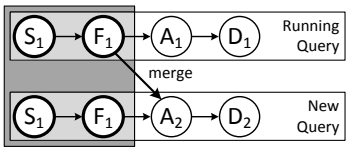


Figure 1: Incremental multi query optimization

2.2 Replication Model

To ensure near zero latency takeover, we assume active replication in the system. Each operator can have multiple replicas exhibiting exact the same behaviour as the primary operator. However, this approach results in the consumption of additional resources.

Query operators and replicas are deployed independently on available hosts. The current placement of operators on hosts is described via a placement function $plc : \mathcal{O} \rightarrow \mathcal{H}$, where $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ is the set of operators for all queries in the system and $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ denotes the set of used hosts.

2.3 Failure Model

According to the query model the system comprises several operators (timed processes) that are executed on a set of hosts. We assume that each host has a probability p to fail with a crash stop failure. Crash stop failures result in an immediate crash of all operators placed on the specific host. Moreover, we assume that Byzantine (value) errors caused by erroneous executions can be transformed into crash stop failures, e. g. by means of Software Encoded Processing [8]. In the following we consider the time period of one day. Due to the fact that in modern data centers the mean time to repair (MTTR) can be up to two days [6], we assume that crashed hosts will stay down for the whole day. Moreover, host crashes are assumed to be uncorrelated, i. e. if some hosts fail others will remain running.

A query is considered broken, iff for at least one of its operators neither a primary operator nor an operator replica is executed anymore.

2.4 Load Model

We assume a load model, which is based on work in the context of the data streaming system Borealis [23] and by Viglas et al. [19]. Each operator has one or two inputs and multiple outputs that are associated with a certain event rate. We assume that the input event rate for source operators is given and the output event rate for source operators equals the input event rate. Each operator *op* has a certain load $load(op)$. A load of 1.0 represents 100 % CPU usage in a fixed time interval - usually one second. This operator load is calculated as product of the input event rate of an operator multiplied by the cost c, which describes the time required by the operator to process a single tuple.

For the example query depicted in Figure 2 the load of the different operators can be specified as $load(S_1) = r_1 * c(S_1)$, $load(F_1) = r_1 * c(F_1)$, $load(A_1) = r_2 * c(A_1)$, $load(D_1) = r_3 * c(D_1)$. Since each operator is associated with a selectivity value, input event rates of operators can be calculated as linear combination of source stream rates and operator selectivity of predecessor operators. For example the input event rate r_3 of operator D_1 can be expressed as $r_3 = sel(A_1) * r_2 = sel(A_1) * (sel(F_1) * r_1)$.

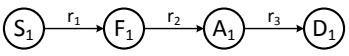


Figure 2: Example query graph

Moreover, each operator has incoming (net_{in}) and outgoing (net_{out}) network traffic. The incoming traffic $net_{in}(op)$ is calculated as product of input rate and input tuple size of the operator, whereas the outgoing traffic $net_{out}(op)$ is calculated by the output rate multiplied with the output tuple size.

For the sake of convenience in the remainder of this paper the term load is used interchangeably with CPU resources.

3. HEURISTIC REPLICATION

Complex event processing systems are exposed to a varying workload caused by different event rates as well as the addition and removal of queries. Figure 3 sketches a fictitious workload of a system processing queries for traffic monitoring. Peak loads indicate rush hour traffic. To deliver results for the queries in real-time, the system has to be equipped with

enough resources to handle those peak loads. However, since the peak load can only be estimated, phases of underprovisioning may occur, which are indicated by the dark gray areas. Moreover, this leads to phases of overprovisioning indicated by the dotted areas.

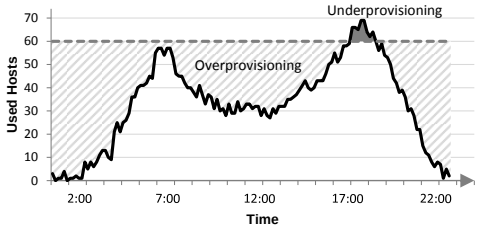


Figure 3: Workload of a data stream processing system

Heuristic replication targets the usage of such spare resources in distributed complex event processing systems to increase the availability of queries. Assume a system that comprises a set of n hosts $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ with an overall CPU capacity of $load_{cap} = n * host_{cap}$. Query operators are placed according to an operator to host mapping plc. The unused CPU resources $load_{rem}$ can be calculated as

$$load_{rem} = load_{cap} - \sum_{op \in \mathcal{O}} load(op)$$

Since queries can be added to and removed from the system at runtime the amount of remaining resources usable to improve the availability of queries in the system changes over time. If the remaining load is smaller than the load of all queries in the system, not all operators can be replicated and, thus, a subset of operators has to be selected for replication. Thus, we propose a three step approach. First, all operators are rated according to their potential to improve query availability by means of an operator importance heuristic (see Section 3.1). Second, the set of operators as well as the replication level, i. e. the number of replica instances, is determined for each operator using an operator selection algorithm (see Section 3.2). Third, the replica instances are deployed on hosts according to an operator placement strategy (see Section 3.3). Finally, in Section 3.4 we present an evaluation.

3.1 Operator Importance Heuristics

The potential of operators to improve the availability of queries is rated using a normalized importance function $imp_{heur} : \mathcal{O} \rightarrow [0, 1]$ that associates each operator in the system with a normalized importance value. Depending on the source of information used for calculating this importance value, static and dynamic heuristics can be distinguished.

3.1.1 Static Heuristics

Static heuristics calculate the importance of operators based on properties observable in the global query graph. Due to the multi query optimization operators can be reused by multiple queries. Hence, if a reused operator fails, all depending queries crash. One possible heuristic, referred to as query out degree heuristic, is based on this observation and rates the importance of operators that are reused by multiple queries more important. The calculation of the query out degree heuristic can be expressed as

$$imp_{QOD}(op) = \frac{deg(op)}{|\mathcal{Q}|}$$

where $deg(op)$ is the number of queries leveraging the operator op, and $|\mathcal{Q}|$ is the number of queries running in the system.

3.1.2 Dynamic Heuristics

Dynamic heuristics assess runtime properties of operators and are, thus, dependent on models providing estimations if an operator is deployed for the first time. If for example the required resources are considered, the heuristics are dependent on the load model of the system (see Section 2.4). One example for a dynamic heuristic is the low utilization heuristic that tries to replicate as much operators as possible. Therefore, operators with a small load are preferred. The calculation of the heuristic can be expressed as follows:

$$imp_{LU}(op) = 1 - \frac{load(op)}{max(load(\mathcal{O}))}$$

where $max(load(\mathcal{O}))$ is the maximum operator load among all operators currently running in the system.

3.1.3 Query Level Heuristics

The heuristics introduced so far operate on an operator level, i. e. operators are treated independent of each other. Hence, operator-level heuristics might cause the selection of only most but not all operators for a query. Not selected operators are weak spots and decrease the achievable availability for the query tremendously. Therefore, we propose to combine operator importance values into query importance values, which are expressed using a normalized importance function for queries $impQ_{heur} : \mathcal{Q} \rightarrow [0, 1]$. Operator selection algorithms operating on query level heuristics will ensure that once a query was selected for replication all operators of that query are replicated. Only if not enough resources are available to replicate a full query, single operators would be selected.

3.2 Operator Selection Strategies

Operator selection strategies take care of the actual operator selection based on the operator importance and the remaining load $load_{rem}$ in the system. The calculated operator selection can be described as function $sel : \mathcal{O} \rightarrow \mathbb{Z}$, which associates each operator with a number of replicas referred to as replication level $rep(op)$.

3.2.1 Simple Operator Selection

A simple operator selection strategy sorts all operators descending based on their relative importance defined via imp_{heur}. Afterwards operators are selected for replication until all spare resources $load_{rem}$ are consumed. Selecting operators for replication includes the determination of the replication level, i. e. the number of replicas to create for a single operator. Different selection procedures can be established:

1. All operators can be selected for replication at least once if enough remaining resources are available. If afterwards spare resources are still available, the replication level for the already selected operators can be

increased stepwise. This procedure ensures that each operator is replicated at least once, if enough resources are available.

2. A replication level larger than one might be set directly for an operator. Hence, some operators might be excluded from replication, if already all resources are consumed.

3.2.2 Optimized Operator Selection

The simple operator selection strategy selects operators for replication based on either a static or dynamic heuristic. Thus, only one type of available information is incorporated at a time and the resulting selection of operators can lead to a non-optimal availability of queries. Therefore, we propose to augment the selection process based on a static heuristic with runtime information by modeling the operator selection problem as 0-1 knapsack problem [16].

Each operator $op_i \in \mathcal{O}$ has a value $v_i = imp_{heur}(op_i)$ and a weight $w_i = load(op_i)$. The maximum weight of the bag equals the remaining resources $load_{rem}$ in the system.

Because the 0-1 knapsack problem is NP-hard and load values of operators and, thus, the optimal solution changes continuously, we suggest an approximation by means of a combination of static and dynamic heuristics. The optimization is based on the intuition that operators with the greatest profit per weight unit have to be selected first. Thus, we propose to leverage a product of the query out degree and low utilization heuristic

$$imp_{QOD*LU}(op) = imp_{QOD}(op) * imp_{LU}(op)$$

as basis for the operator selection.

3.3 Fault-tolerant Operator Placement

Operators selected for replication via the operator selection algorithm have to be deployed on hosts. However, depending on the chosen operator placement for the same set of selected operators different availabilities can result.

3.3.1 Simple Bin Packing

A bin packing algorithm [4] for fault-tolerant operator placement minimizes the number of used hosts, so that idling hosts can be turned off to save energy. This property is in line with the notion of elasticity. We propose a bin packing algorithm, which is an extended version of a first-fit bin packing, which has a complexity of $O(mn)$, where m is the number of replicas that shall be placed and n the number of hosts.

Each host is modeled as bin, where the available CPU resources form the capacity. Replicas are first sorted in decreasing order according to their normalized importance and then assigned using their load as weight. Moreover, two constraints are ensured: (i) the network capacity of a host should not be exceeded, (ii) two replicas of the same operator are never placed on the same host. To reduce the consumed network bandwidth, possible hosts are ordered according to a neighboring factor representing the amount of predecessor and successor operators deployed on the same host.

3.3.2 Colored Bin Packing

Even though the simple bin packing algorithm is replica-aware due to the additional placement constraint, the system availability is not maximized because as many replicas of different operators as possible are placed on one host. Thus, multiple replicas crash in case of a host failure.

Therefore, we propose to use a bin packing algorithm with color constraints [5]. Besides load and network consumption, each replica will be associated with a unique color. The colored bin packing algorithm ensures that each host contains only replicas with at most c distinct colors, where c is a pre-defined positive integer. Because the number of replicas placed on the same host shall be minimized, an upper bound C with $c \leq C$ is specified for the color constraint. Then the placement problem can be formulated with an additional constraint that strives to minimize the parameter c. If no placement can be found without violating the upper bound C, the simple bin packing algorithm is used to calculate a placement.

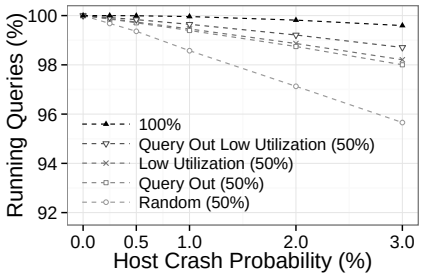


Figure 4: Comparison of query availability for different heuristics

3.4 Evaluation

We have performed a preliminary simulation-based evaluation of the proposed approaches. We used a query generator to generate queries based on six query templates that differ in structure and operator parameters. Operators were deployed on simulated hosts with a CPU capacity of $load_{cap} = 1$. We assume a system comprising 100 hosts. Moreover, operators were replicated at most once. To guarantee statistical correctness, 1000 runs were conducted for each experiment and values have been averaged.

Figure 4 depicts the percentage of remaining running queries after a time period of one day assuming different host failure probabilities $p = \{0.0025; 0.005; 0.01; 0.02; 0.03\}$ and different heuristics for rating the importance of operators. Because a constant system load $load(\mathcal{O}) = 100$ was generated, the stated percentages of resources available for replication are equal to the actual remaining load $load_{rem}$ in the system. Leveraging either the low utilization heuristic or the query out degree heuristic to select operators for replication improves the percentage of remaining running queries by approximately 1.7 percentage points compared to a random operator selection assuming a host failure probability of $p = 0.02$ and $load_{rem} = 50$. Using a combination of both heuristics improves the query availability further by 0.4 percentage points, resulting in 99.2 % remaining running queries.

Figure 5 shows the availability increase as a function of available resources for replication. A combination of query out degree and low utilization heuristic results in 99.2 % remaining running queries, if a host failure probability of $p = 0.02$ and $load_{rem} = 50$ is assumed. Moreover, the achievable percentage of running queries is only diminished

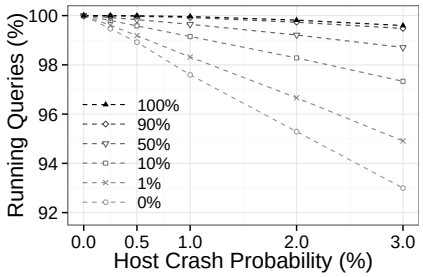


Figure 5: Query availability for combination of query out degree and low utilization heuristic

by approximately one percentage point compared to full replication, if 50 % less spare resources are used for replication and the host failure probability is $p = 0.03$. This result can be explained by the fact that with $load_{rem} = 50$ available for replication still 80 % of the query operators are replicated. Moreover, those operators are reused to a large extent or have small cost.

4. REPLICATION OPTIMIZER

Service level agreements are important in scenarios where a certain availability is required. However, achieving a higher availability by replicating more operators requires more resources. Therefore, the solicited availability and the resulting resource consumption should be traded off. Replication of operators in a system can be optimized with respect to two different optimization goals:

1. For a given resource limit $cost_{max}$, the availability of queries $A(\mathcal{Q})$ can be maximized.
2. For a given availability of queries $A(\mathcal{Q})$, the resource consumption $cost(\mathcal{Q})$ can be minimized.

The achieved availability of queries is influenced by the set of operators selected for replication, the number of replicas that is created for each of the selected operators, as well as the placement of replicas on hosts. All those influential parameters are reflected in the placement *plc* and, thus, the placement can serve as predictor for the achievable availability. To optimize the replication decisions according to the two specified optimization goals, additional models have to be incorporated. To estimate the costs for a placement, the load model for query operators can be leveraged. Moreover, an availability model is required that enables the estimation of the query availability resulting from a certain placement.

Assuming the availability of those models, the optimization problems can be formulated as follows:

$$max\{A(plc) \mid plc \in \mathcal{P}; cost(plc) \leq cost_{max}\}$$

and

$$min\{cost(plc) \mid plc \in \mathcal{P}; A(plc) \geq A(\mathcal{Q})\}$$

where $\mathcal{P}$ is the set of all possible placements resulting from different operator selections, different replication levels, as well as different placement strategies.

Those optimization problems can be solved using well-known optimization algorithms like genetic search [10]. However, to lower the effort for the optimization, heuristics to restrict the search space have to be developed.

5. DYNAMIC REPLICATION

Elastic data stream processing systems are able to cope with varying query as well as event load. Mechanisms to dynamically allocate and release hosts depending on the actual demand prevent costly overprovisioning and performance barriers due to underprovisioning. Ideally the system can scale out indefinitely to serve high event rates and scale in to lower the used resources in case of low utilization.

Processing queries in an elastic environment poses various challenges. Once new queries are submitted to the system, the encompassed operators have to be distributed to the running hosts. Because of operator reuse the load of already deployed operators changes. To not impair the performance of the system two reactive actions are taken: (i) overloaded hosts might be relieved by automatically migrating operators from one host to another, (ii) overload situations that cannot be resolved by moving operators from one host to another are resolved by splitting the operator into several operator instances that handle only a portion of the overall load and can be deployed independently.

However, handling replicas in a dynamic system is demanding:

- The complexity of the reconfiguration caused by the exoneration of overloaded hosts is increased since additional communication channels for replicas have to be maintained.
- If operators are split into several instances, all replicas have to be split too in order to maintain a consistent system state.
- Systems that are used in conjunction with heuristic replication (see Section 3), have to decide in case of spare resources whether to release hosts or to create additional replicas.
- Reconfiguration routines have to ensure that the new placement does not violate existing service level constraints for the queries (see Section 4).

6. RELATED WORK

Fault tolerance techniques for data stream processing systems like active replication [2], check pointing [14], and a combination of active and passive replication [24], are key enablers for our proposed approaches.

Moreover, operator placement algorithms are leveraged, which have been studies extensively by various authors. A survey is available in [15]. Repantis et al. [18] presented a procedure for replica placement considering performance constraints like end-to-end latency. The ZEN system [3] models different levels of availability in a systems and tries to assign most important operators to hosts with the highest availability. Another replication scheme based on graph coloring is presented in [22].

Optimization in the area of data stream processing systems is done for example to achieve an optimal overall utilization [21], or optimal utilization with limited resources [13].

An approach related to that in this paper has been studied by Fernandez et al. in [7]. The authors present an integrated approach to scale out streaming systems while achieving fault tolerance via check pointing. In this paper we focus, however, on active replication to ensure minimal latency and envision also scale in.

7. SUMMARY

As of today, many data stream processing systems use replication to ensure high availability in case of host failures. However, to replicate a system completely, at least twice the resources have to be allocated, which is costly.

We proposed a heuristic replication approach enabling the use of remaining system resources to increase the availability of queries. An operator selection algorithm is used to determine a subset of operators for replication that are then placed via an operator placement algorithm. Moreover, we suggested a replication optimizer which allows users to guide the replication while trading off cost and availability. Finally, the challenges arising from a combination of these techniques with elastic data stream processing systems were discussed. Using heuristic replication as well as the replication optimizer with a system reacting on changes, e. g. in event rate and selectivities, demands the adaptation of the placement routines. However, the normalized importance functions as well as the optimization routines might be reused unchanged.

To validate the approaches we simulated a heuristic replication approach comprising operator selection as well as operator placement strategies. Given a set of remaining resources, the fault tolerance of complex event processing systems is improved. Choosing a proper heuristic can improve the availability two percentage points compared to a random operator selection. Compared to full replication only one percentage point is lost spending 50 % less resources for replication.

In the future, we plan to implement the proposed approaches with a state-of-the-art streaming system.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), 2008.

[3] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. S. Turaga, and C. Venkatramani. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM*, pages 1319–1327, 2008.

[4] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for np-hard problems. chapter Approximation algorithms for bin packing: a survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[5] M. Dawande, J. Kalagnanam, and J. Sethuraman. Variable sized bin packing with color constraints. *Electronic Notes in Discrete Mathematics*, 7:154–157, 2001.

[6] J. Dean. Handling Large Datasets at Google: Current Systems and Future Directions. In *Data-Intensive Computing Symposium*, 2008.

[7] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM International Conference on Management of Data (SIGMOD)*, New York, NY, 06/2013 2013. ACM, ACM.

[8] C. Fetzer, U. Schiffel, and M. Süßkraut. An-encoding compiler: Building safety-critical systems with commodity hardware. In *SAFECOMP*, pages 283–296, 2009.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *SIGMOD Conference*, pages 1123–1134, 2008.

[10] D. E. Goldberg. *Genetic Algorithms*. Pearson Education, 2013.

[11] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. B. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE*, pages 176–185, 2007.

[12] C. Jin and J. G. Carbonell. Predicate Indexing for Incremental Multi-Query Optimization. In *ISMIS*, pages 339–350, 2008.

[13] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. Sqpr: Stream query planning with reuse. In *ICDE*, pages 840–851, 2011.

[14] Y. Kwon, M. Balazinska, and A. G. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *PVLDB*, 1(1):574–585, 2008.

[15] G. T. Lakshmanan, Y. Li, and R. E. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.

[16] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Wiley-Interscience series in discrete mathematics and optimization. J. Wiley & Sons, 1990.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDM Workshops*, pages 170–177, 2010.

[18] T. Repantis and V. Kalogeraki. Replica placement for high availability in distributed stream processing systems. In *DEBS*, pages 181–192, 2008.

[19] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, pages 37–48, 2002.

[20] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *SoCC*, pages 193–204, 2010.

[21] J. L. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.

[22] F. Xiao, T. Kitasuka, and M. Aritsugi. Economical and fault-tolerant load balancing in distributed stream processing systems. *IEICE Transactions*, 95-D(4):1062–1073, 2012.

[23] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.

[24] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. A hybrid approach to high availability in stream processing systems. In *ICDCS*, pages 138–148, 2010.

Dynamic Partitioning of Big Hierarchical Graphs*

Vasilis Spyropoulos
Athens University of Economics and Business
76 Patission Street
Athens, Greece
vasspyrop@aueb.gr

Yannis Kotidis
Athens University of Economics and Business
76 Patission Street
Athens, Greece
kotidis@aueb.gr

ABSTRACT

Hierarchical graphs are multigraphs, which have as vertices the leaf nodes of a tree that lays out a hierarchy, and as edges the interactions between the entities represented by these nodes. In this paper we deal with the management of records that are the edges of such a graph by describing a model that fits well in a number of applications, many of which deal with very big volumes of streaming distributed data that have to be stored in a way so as their future retrieval and analysis will be efficient. We formally define a partitioning schema that respects the hierarchy tree, and apply these ideas by using well known open source big data tools such as Apache Hadoop and HBase on a small cluster. We built a framework on which we examine some basic policies for the partitioning of such graphs and draw interesting conclusions regarding the quality of the partitions produced and their effectiveness in processing analytical queries drawn from the imposed hierarchy.

1. INTRODUCTION

There are numerous applications such as management and visualization of Telecommunications data [1], Web log mining [2] or Internet traffic analysis [3], in which data records can be described as edges between vertices of a *hierarchical graph*, i.e a directed multigraph whose vertices are also the leaf nodes in a hierarchy tree. As an example, Call Detail Records (CDRs) can be naturally depicted via a massive graph structure in which nodes represent customers' phone numbers and edges between them their calls. At the same time, the nodes of this graph are the leaves of a tree that indicates their location and superimposes a geographical hierarchy over this data [4].

You can see such an example in Figure 1 which presents a small part of the hierarchy of locations in Greece. In this Figure, Attiki and Messinia are states of Greece, while Athens, Piraeus and Kalamata are cities in these states. The unlabeled nodes represent subscribers of a telephony network. Then, each call serviced by the telephony network instantiates a new directed edge in the graph, between the respective vertices (caller and callee) located at the leaves of the hierarchy tree, as is shown in the Figure. This hierarchy is exploited in order to help pose queries that seek to retrieve certain records for further analysis. For instance in order to calculate statistics on the out-of-state calls originating in Athens, this intention may be described by a query edge between Athens and Greece in the tree. Similarly, query edge (Attiki, Attiki) denotes the set of calls originating and terminating within this state. The aforementioned query edges are shown in Figure 1 (dashed lines). Another example where similar hierarchical graphs exist, is social networks where users are organized in groups according to their location or other characteristics such as age or interests and we need to record the interactions between them. Users (e.g. analysts) of such data often need to answer queries regarding interactions or distributions of records between hierarchy groups not necessarily belonging to the same level of the hierarchy.

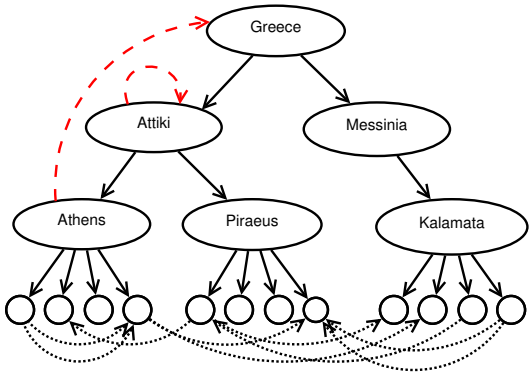


Figure 1: Example of a hierarchical graph - the graph consists of the unlabeled nodes, which are also the leaves of a hierarchy tree, and the edges/interactions between them

In the aforementioned applications, this kind of graphs can grow to enormous size. For instance a large telecom provider may service hundreds of millions of calls per day, each triggering a new edge in the graph. Moreover, this data is distributed by nature as it is being streamed from distant locations (e.g. call centers, web hosts, ip routers). Thus, we need solutions that can cope with the volume, but also with the streaming and distributed nature that characterize this kind of data.

In our work we address these challenges by moving the storage and processing of these graphs to the cloud. We propose a system that uses the distributed data store HBase [5] running on the

*This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: RECOST

Hadoop distributed file system [6], but also MapReduce [7] techniques so as to handle a continuous stream of updates efficiently. Our system leverages the available degree of hardware parallelism by devising a dynamic partitioning scheme over the streamed edges of the hierarchical graph. Our techniques aim at generating partitions that correspond to clusters of graph edges, which are naturally mapped to collections of nodes in the hierarchy tree, while respecting the distribution of the streamed records. In this way, analysis of the records based on the superimposed hierarchy can be performed in an efficient manner. Our contributions are:

- We revisit the problem of managing massive hierarchical graphs that are streamed by many applications of interest. Our techniques utilize emerging computational and data management platforms for manipulating large, dynamic and distributed collections of records in a cluster of machines. Available parallelism is exploited via a dynamic partitioning scheme we propose for the streamed records.
- We formally define the space of choices for partitioning the streamed graph, while respecting the hierarchy tree that is superimposed over its nodes. We then present a number of interesting partitioning policies, and describe the details of the system we built for implementing our framework while utilizing off-the-shelf tools.
- We present an experimental evaluation of our system using a small cluster of machines. Our results demonstrate the efficiency of our system in managing massive graphs scaling to millions of edges. We also provide a comparison among the partitioning policies we implemented based on the results of a number of experiments that we conducted.

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we formally introduce our framework, discuss the type of graph data and queries we consider. Then, we describe a partitioning scheme based on the tree hierarchy that accompanies the graph data, a number of partitioning policies that we implemented and discuss the architecture of our system. Section 4 presents our experiments and Section 5 contains concluding remarks.

2. RELATED WORK

Interest in graphs and their applications in data management has been renewed due to the wide spread of fields such as social networks and the semantic web. In the same time there is a profound need for the efficient management of big distributed data. As a result we can see a lot of recent work done in the area, ranging from graph databases to distributed graph processing or graph partitioning techniques. The latter mainly cope with the problem of splitting a large graph by assigning its vertices into independent partitions. While there are several variations of the problem, a typical objective is to obtain partitions such that the sum of the vertex weights across partitions is even while the sum of the inter-partition edges is minimized [8, 9]. The work in [10] proposes data partitioning that is guided by the user's queries. Another approach that aims at the partitioning of graphs across clusters of servers in a dynamic way by using queries during the runtime of the system can be found in [11]. Our query-driven partitioning policy described in Section 3 is motivated by these ideas but the actual setting is different. In [12] the authors present SPAR, a social partitioning and replication middle-ware that uses the social graph structure in order to achieve data locality. In our work we also use a structure to guide the partitioning but the structure we use is a hierarchy tree.

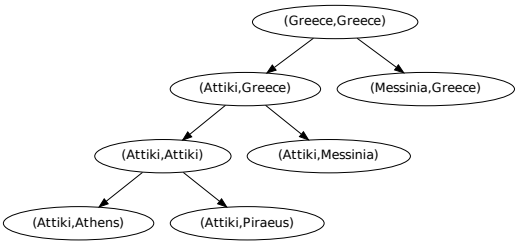


Figure 2: Example state of a partition tree T_P

The objective in our work is quite different to vertex partitioning since we actually do partitioning of the edges of a hierarchical multigraph. These edges represent interactions between nodes that need to be investigated according to the imposed hierarchy. In a typical scenario aggregation of these edges at the higher levels of the hierarchy tree is more important for the application, while in certain applications, such as analysis of call detail records, decision making based on fine-granularity statistics (i.e. low-level aggregations) is in-fact prohibited by law, so that certain carriers cannot obtain unfair advantage over their competitors. Our techniques can benefit systems built for visualizing hierarchical multigraphs (e.g. [1]). Moreover, indexing techniques for hierarchical multigraphs such as [4] and multi-dimensional indexes over key-value stores as in [13] can be incorporated in our system for providing fast access to individual records within the partitions created by our framework.

3. SYSTEM OVERVIEW

3.1 Definitions

Assuming a rooted hierarchy tree T, we denote the set of its leaves as $Leaves(T)$. We refer to a subtree rooted at a vertex $x \in T$ as T_x. For vertices u and v of T we say that v is *descendant* of u if there is a path descending from u to v, including the case that u equals v. Vertices u and v are called *comparable* in T if one of them is descendant of the other and *incomparable* in T if neither u nor v is a descendant of the other. Also, by $depth(u)$ we shall refer to the number of edges that have to be crossed so as to get from u to the tree root node.

A *hierarchical graph* is a multigraph $G(T, V, E)$, where T is a tree, V is the set of vertices in the graph, which are also the leaves of T (V=$Leaves(T)$) and E is a multiset of edges between the nodes in V. We assume that each instance of an edge is labeled with a unique identifier in order to be able to assign data on them (i.e. in the CDRs example, each edge is associated with a unique key that identifies the corresponding CDR).

In order to represent the set of edges between nodes in T and their relationships, we define the graph G_{T^2}. Each possible pair of nodes u and v in T is a vertex $[u, v]$ in G_{T^2} and we refer to u as the source node in T and v as the destination node in T. The edges in G_{T^2} imply a hierarchy inherited from the hierarchy described by tree T. In particular, there is an edge between nodes $[u_1, v_1]$ and $[u_2, v_2]$ in G_{T^2} if exactly one of the following conditions hold: (i) u_2 is a child of u_1 in T, (ii) v_2 is a child of v_1 in T.

Graph G_{T^2} is used in our framework in order to define our partitioning scheme on the edges of the hierarchical graph. Moreover, the nodes of G_{T^2}, as will be explained in Section 3.2, are used in order to model possible queries on this data. Then, the edges of G_{T^2} will determine partitions that contain relevant data for a query.

Each vertex $[u, v] \in G_{T^2}$ is a candidate partition $P_{u,v}$ to be ma-

terialized by our partitioning scheme. Keeping all G_{T^2} in memory is not a feasible solution at its size is quadratic on the size of T. As will be explained, our partitioning process progressively splits the hierarchical graph and constructs a *partition tree* T_P, which is a subgraph of G_{T^2}. When we need to locate partitions so as to insert new records or answer queries we use T_P as our lookup structure. An example partition tree T_P which follows the hierarchy of Figure 1 is shown in Figure 2

Finally, by $C_{u,v}$ we refer to a counter of how many records are contained in the partition $P_{u,v}$ and by $threshold$ to the value that when the number of records in a partition grows bigger than, the partition has to be split in smaller ones and then get dropped.

3.2 Hierarchical Queries

In our framework, retrieval of edges belonging to the hierarchical graph is accomplished via queries that are modeled using the hierarchy tree T. In particular a query $Q_{u',v'}$ is denoted as a query edge in T (see Figure 1). This query denotes our intention to retrieve all edges that have as source vertices the leaves of $T_{u'}$ and as destination vertices the leaves of $T_{v'}$. When such a query arrives we need to be able to decide which of the materialized partitions in T_P may contain relevant graph edges. In order to achieve that we traverse T_P in a top-down fashion and check each vertex $P_{u,v}$ against the query $Q_{u',v'}$. The partition is considered useful in answering the query when their respective source and destination vertices are comparable in T. We continue traversing T_P descending the useful partitions until we get to the active partitions (active partitions are these that contain data and are pointed by the leaf nodes of T_P as explained in Section 3.3), which are returned to the query for further processing (e.g. filtering of relevant edges). In case that during the traversal we come across a $P_{u,v}$ for which it stands that u is equal to u' and v is equal to v' then we stop the traversal and retrieve the leaf nodes in the subtree rooted at $P_{u,v}$. In that case, all the edges in the respective partitions are returned to the user.

In our running example, assuming that the partition tree T_P is at the state shown in Figure 2 and that we have to answer query $Q_{Athens,Messinia}$ that retrieves all CDRs from locations in Athens to locations in the state of Messinia, we first check the root of T_P which is [Greece, Greece]. Since Athens is comparable to Greece and Messinia is comparable to Greece we continue with examining the children of [Greece, Greece], which are [Attiki, Greece] and [Messinia, Greece]. Athens is comparable to Attiki and so is Messinia to Greece, so node [Attiki, Greece] is useful, but Athens is incomparable to Messinia so we do not have to further investigate the node [Messinia, Greece] or any node in $T_{Messinia,Greece}$. Next we have to check nodes [Attiki, Attiki] and [Attiki, Messinia] which are the children of [Attiki, Greece]. Athens is comparable to Attiki but Messinia is incomparable to Attiki so [Attiki, Attiki] is not considered useful. On the other hand [Attiki, Messinia] is useful since Athens is comparable to Attiki and Messinia is comparable to Messinia. [Attiki, Messinia] is a leaf node in T_P and so the traversal ends here and the partition pointed by [Attiki, Messinia] is the result returned to the query.

3.3 Overview of the Partitioning Process

A high level description of the partitioning process is as follows: In the beginning let T_P consist of just one vertex $[r, r]$ where r refers to the root of T. That means that we initially materialize just one global partition $P_{r,r}$ containing all possible edges amongst leaves in T. When $C_{r,r}$ grows greater than $threshold$ the split process is triggered. The split process decides whether $P_{r,r}$ will be split by its source or destination node, depending on the rules of the chosen partitioning policy, which is discussed later. Each outcome is encoded by a set of nodes that are reachable from node $[r, r]$ in graph G_{T^2}, depending whether the respective edge denotes a parent-child relationship on the source or destination node.

After the split, the vertices representing the new partitions are added to T_P. Vertex $[r, r]$ in T_P points no longer to an active partition but we keep it since it describes the records contained in the active partitions pointed by its descendants (the new vertices that we added) and we use this information when we traverse T_P in order to insert new records or answer a query. This process takes place for every active partition $P_{u,v}$ when $C_{u,v}$ grows greater than $threshold$ after the insertion of new records. This way the vertices in T_P that point, or previously have been pointing, to an active partition, form a hierarchical tree. At any moment the leaves of T_P point to the active partitions while the inner nodes, including the root, are "aggregations" of these partitions.

Any node in T_P can optionally maintain a series of useful application specific statistics such as the number of records in the partition, aggregations over measures of these records, calculations regarding heavy hitters such as top-k sources and top-k destinations, etc. Furthermore, when fast approximate answers are desired by the application (for example during exploratory data analysis or as a preview while the exact answer is computed) it is also possible to maintain synopses such as Sketches [14], Histograms [15, 16] or Wavelets [17, 18] on the nodes of T_P. Since these nodes are traversed while new data is added in the partitions, maintenance of these synopses can be easily incorporated in the process. While these extensions are applicable in our framework, their discussion is beyond the scope of this paper.

In what follows we describe the different split/partitioning policies that we implemented in our system and used in our experiments. First we describe two simple query agnostic policies and next, in more detail, a partitioning policy that we call Query-Driven Partitioning that decides the split to materialize by taking under consideration a set of queries that are most important to the user and makes the split decisions according to them.

Query-Agnostic Policies: The first two policies assume no previous knowledge about the interests of the user. Each of them though utilizes a different heuristic as explained below.

- **Round-Robin Partitioning:** Round-Robin is a simple approach to partitioning the hierarchical graph. Partitions in T_P that need to split, are split alternately by source or destination. This process results in creating balanced partitions in the sense that source and destination nodes in a partition have a maximum distance of one hierarchy level. That way the partitions created are not biased towards the source or destination nodes of the constituent edges.
- **Min-Split Partitioning:** Min-Split partitioning policy is a heuristic method, which tries to create the minimum number of new partitions, when an active partition overflows. This policy seems preferable when the goal is to create as few active partitions as possible, while keeping their size close to the selected threshold. Thus, when it has to make a split choice, it simply chooses between the candidate splits the one containing fewer partitions.

Query-Driven Partitioning: In Query-Driven Partitioning we assume that we have a prior knowledge of the queries that the users of the system are mostly interested in. So, when we have to make a split choice we chose the candidate split that suggests a partitioning better suited to answer the set of queries. In what follows we present the idea of Query-Driven Partitioning for hierarchical graphs in a formal way.

Let $P_{u,v}$ be a partition in the partition tree T_P and $Q_{u',v'}$ be a query asking for all records having as source and destination all the nodes that are leaves of the hierarchy tree's T subtrees $T_{u'}$ and $T_{v'}$, respectively. Recall that a partition is useful for answering a query if their respective source and destination nodes are comparable in T, otherwise the partition is pruned while navigating the partition tree in search of answers to the query.

For a useful partition, we can define a measure of the overhead that the retrieval of $P_{u,v}$ adds to the overall cost of answering the query by considering the number of records that belong to $P_{u,v}$ but are not part of the result of $Q_{u',v'}$. We can use this measure to make the split choice for a partition to be split, by calculating the query answering overhead for each of the candidate splits. Extending this, we can calculate the overhead not just for one query but for a set of queries that the users are mostly interested in.

In order to measure the overhead of a useful partition for a given query, we have to estimate the portion of the partition records that are not useful for the query, but will have to be retrieved when scanning the partition for relevant data. Since both the partition and the query follow the hierarchy implied by the hierarchy tree T we should check the partition's source and destination nodes u and v against the query's source and destination nodes u' and v', respectively. We will describe the procedure for u and u', but whatever we mention holds true also for v an v'.

Since nodes u and u' are comparable (otherwise the partition is not useful), we have to consider the following three cases: (i) u equals u', (ii) u is a descendant of u' ($depth(u) > depth(u')$, and (iii) u' is a descendant of u ($depth(u) < depth(u')$). Let $fitness(u', u)$ denote the portion of the leaf nodes in T_u that are also leaves in the subtree of $T_{u'}$. In the first and second cases we can safely infer that $fitness(u', u)$ equals to 1, since T_u is contained in $T_{u'}$. On the contrary, in the third case, $T_{u'}$ is contained in T_u and, thus, $fitness(u', u)$ is calculated by considering the ratio of the leaves of $T_{u'}$ over the leaves of T_u.

$$fitness(u', u) = \begin{cases} 0 & \text{, if } u \text{ and } u' \text{ are not comparable} \\ 1 & \text{, if } u \text{ and } u' \text{ are comparable and } depth(u) \geq depth(u') \\ \frac{|leaves(T_{u'})|}{|leaves(T_u)|} & \text{, otherwise} \end{cases}$$

Then, the fitness of the partition for the query is computed as:

$$fitness(Q_{u',v'}, P_{u,v}) = fitness(u', u) \cdot fitness(v', v)$$

Intuitively, this measure estimates the percentage of records in the partition that are useful for the query, assuming no additional knowledge on the data distribution is given.

In case the partition is split into k sub-partitions, assuming a uniform distribution of the records in $P_{u,v}$, then each of these partitions will receive $\frac{C_{u,v}}{k}$ of records, where $C_{u,v}$ is the size of the partition. Then, given that we have calculated the fitness for each of these smaller partitions denoted as $f_1, \ldots f_k$, respectively, we compute the *overhead* of the split as the number of non-useful to the query records expected to be retrieved from the set of partitions belonging to the candidate split as:

$$overhead(Q_{u',v'}, Split(P_{u,v})) = \frac{C_{u,v}}{k} \sum_{i=1\ldots k, f_i>0} (1 - f_i)$$

For a set of queries the cumulative overhead of the split is computed by summing the estimated overhead for each query. Thus, given a choice of splitting the partition by source of destination, we compare the overheads that we calculated for each of the splits and select to materialize the one with the lowest number. As have been explained, our calculations are based on the assumption that the data distribution within the partition is uniform. Of course, this is a bold assumption and we expect that the system may occasionally make wrong decisions, leading to suboptimal splits. An easy workaround is to consider additional statistics, for instance in the form of sketches, that will help better estimate the distribution of records within a partition, at the cost of increased overhead due to bookkeeping of these synopses. We leave exploration of such choices as future work.

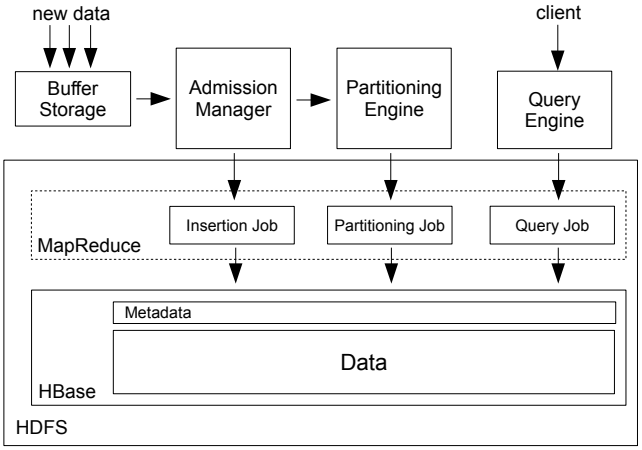


Figure 3: Framework overview

3.4 Implementation Overview

We implemented our system as a framework consisting of four main modules, the Buffer Storage, the Admission Manager, the Partitioning Engine and the Query Engine. It uses the well-known distributed data store HBase running on top of HDFS (the Hadoop Distributed File System) and also the Hadoop MapReduce engine in order to accomplish tasks such as loading and repartitioning of the data. We materialize each partition as an HBase table so as to make easier the retrieval of records belonging to one partition by involving the scan of one and only table. Another choice would be to use one big table to store all the records and define each partition space by applying a compound row-key design.

Figure 3 provides a high-level view of the framework. The Buffer Storage module is located at the input of the system and stores temporarily the new records that are streamed from distributed sources, into text files. When the size of these records grows bigger than a maximum buffer size, then the Buffer Storage module sends a message to the Admission Manager module, which executes the task of pulling the records from Buffer Storage and loading them into HBase. After each load of new records, Admission Manager checks for partitions that grew over the partitioning threshold. For any such partition, Admission Manager passes the required instructions to Partitioning Engine, which decides a split according to the selected policy amongst the ones described earlier and applies it.

For the support of these tasks, the system maintains a set of metadata, such as the hierarchy and partition trees that are used, updated and shared by all the different modules. The actual execution of the tasks is taking place as a number of MapReduce jobs, with the most important of them being the Insertion Job which puts each new record in the appropriate partition by looking up the hierarchy and partition trees, and the Partitioning Job which moves the data from a splitting partition to the new partitions.

Finally, we also implemented a Query Engine, which accepts a

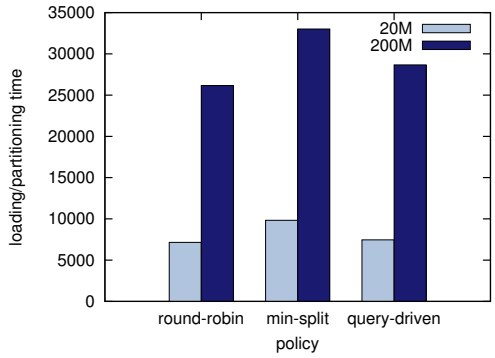


Figure 4: 20M vs 200M Records Loading Times

user query or set of queries and, by looking up the hierarchy and partition trees, discovers the partitions that possibly contain relevant records, scans them and returns the results to the user.

4. EXPERIMENTS

4.1 Experimental Setup

All experiments were conducted on a cluster of seven virtual machines located at the GRNET's cloud infrastructure Okeanos[1]. Each machine had 4 processors, 4 GB of memory, while the disk sizes varied from 40 to 100 GB. The operating system installed was Ubuntu Server 12.04 and we used Hadoop version 1.1.2 and HBase version 0.94.6.1. We used one machine as the system master running the NameNode, JobTracker and HMaster daemons, while each of the remaining six slave machines were running instances of the DataNode, TaskTracker and HRegionServer daemons.

In order to be able to generate massive graph data records, we wrote a custom CDR data generator. We used the geopolitical hierarchy of Greece as described by the Ministry of Interior[2] to create the hierarchy tree and then we generated a number of phone numbers for each city in it. The resulting hierarchy tree consisted of five levels, which from top to bottom are Country, District, State, City and Phone Number, each of them having a node count of 1, 13, 58, 324 and 1134698 nodes, respectively. The phone numbers consist of regular phone numbers that make and receive calls, inbound-only phones as those found in customer service departments or telemarketing, and outbound-only phones as those used by marketing agencies. Each record in the experiment data sets consists of a source and destination phone number and a unique call record id. Last, we created a random set of 10 queries which we used to guide the Query-Driven policy, but also to examine the performance of each policy in answering them. The queries were picked randomly amongst all possible queries having as source or destination inner nodes of the hierarchy tree, meaning we excluded the root (country) and the leaves (single phone numbers).

4.2 Loading/Partitioning Time

The first experiment we conducted is a comparison of the loading times for each of the partitioning policies we implemented. We created a data set of a total of 50 million CDR records and broke it into an initial ingest set of 20 million records and 10 append sets of 3 million records each. For each of these 11 steps we traced the total time it took to insert the records in HBase and perform the repartitioning of the schema, when needed. The results of this experiment are summarized in Figure 5. We can see that the Min-Split and Query-Driven policies spent slightly more time in repartitioning the hierarchical graph. This is explained since Min-Split follows a conservative approach of repartitioning the data by taking the minimum number of splits at each step, resulting in more subsequent splits. The Query-Driven policy is often keen to repartition the data by extending the partitioning tree in order to best fit the input queries that drive its selection process. Finally, in Figure 4 we compare the scalability of our framework using a larger number of input data (200 million records) for each policy. Compared to the smaller dataset, the Figure suggest a sublinear increase of the loading times, something that is contributed to the overhead times of the underlying frameworks (e.g. MapReduce jobs setup) that affects more (proportionally) the smaller input.

[1]https://okeanos.grnet.gr/
[2]http://www.ypes.gr/

4.3 Partitions' Quality

In order to examine the quality of the partitions created by each of the policies tested, we propose a measure that we call $distance$. This metric can be used in applications where the goal is to derive the fewer number of partitions that are each smaller or equal to the selected threshold. Given the actual final partition sizes $size_i$, where i=1 . . . p_m and p_m is the number of the active partitions for policy m at the end of the loading phase, we define $distance_m$ to be:

$$distance_m = \sqrt{\sum_{i=1}^{p_m} (size_i - threshold)^2}$$

Intuitively, a smaller distance value denotes a set of partitions that are created evenly near, but not exceeding, the selected threshold. We have calculated and present the value of the distance metric for each policy and after each loading/partitioning job in Figure 6(a).

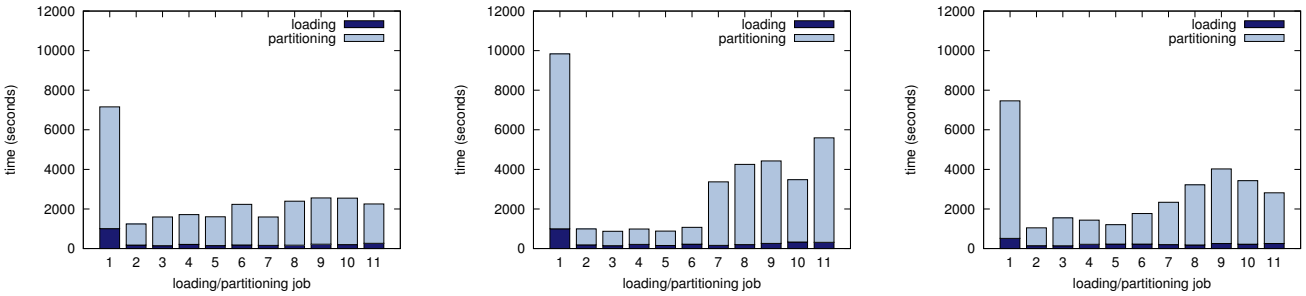
What is worth noticing in this Figure is that the Min-Split policy, while in the beginning was the best amongst the others, later on it created partitions that had a larger collective distance. This is explained by the fact that the choice of the smaller split, leads to many deep splits of the partition tree and, subsequently, when the leaves are reached, the deep partitions that need to be split are getting split in high levels of the hierarchy (on the opposite direction) since this is the only feasible split left. This fact also leads to the increase of the number of active partitions that is evident in Figure 6(b). Thus, even though at each step Min-Split makes (locally) optimal decisions regarding the split that leads to the smaller increase of the distance metric, the final resulting partitioning is worse that the ones achieved by the other policies.

4.4 Queries Answering

In order to examine the effectiveness of the dynamic partitioning schema, for each of the policies we ran the set of queries mentioned in Section 4.1. We used these same queries to guide the Query-Driven policy. In Figure 6(c) we can see a comparison of the total records retrieved by each of the policies, and the fraction of them that were useful in answering these queries. As expected the Query-Driven policy has better precision than the other two policies, while the Min-Split policy, which has gone deep in the hierarchy tree, was not able to create partitions suitable for the selected set of random queries.

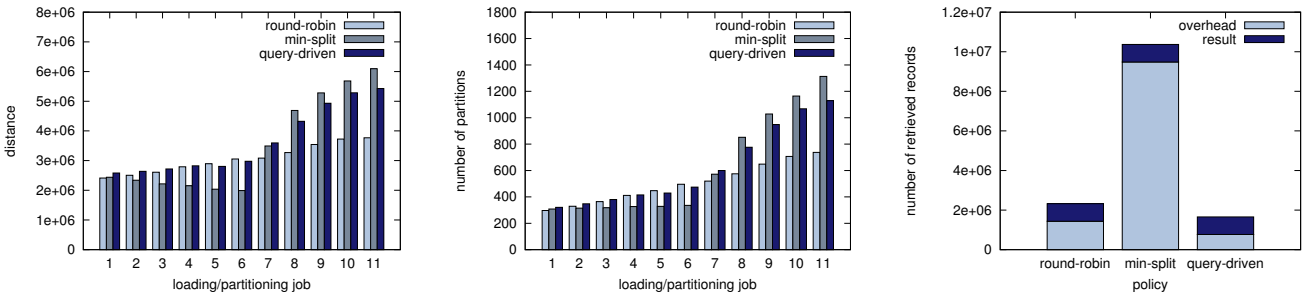
5. CONCLUSIONS

In this paper we considered the problem of managing big hierarchical graphs by exploiting the implied hierarchy so as to partition the data edges in a way that would better support future retrieval



(a) Round-Robin (b) Min-Split (c) Query-Driven

Figure 5: Policies' loading and partitioning times



(a) Partitions' distance (b) Number of active partitions (c) Queries

Figure 6: Comparison between policies

and analysis. We evaluated a number of dynamic partitioning policies using open source big data tools on a small cluster of nodes. From the policies considered, the Query-Driven partitioning policy lead to partition schemes that enable faster analysis of the records, assuming that some a-priori knowledge of the user queries is given. In an uncertain environment, the Round-Robin policy seems to result in more balanced partitions with good query performance. Moreover, it has been shown to have the best performance in the loading and partitioning processes.

6. REFERENCES

[1] J. Abello and J. Korn, "Visualizing massive multi-digraphs," in *Proceedings of INFOVIS*, pp. 39–47, 2000.

[2] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Comput. Netw.*, vol. 33, June 2000.

[3] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 251–262, Aug. 1999.

[4] J. Abello and Y. Kotidis, "Hierarchical graph indexing," *Proceedings of the twelfth international conference on Information and knowledge management*, 2003.

[5] "Apache HBase." `http://hbase.apache.org/`.

[6] "Apache Hadoop." `http://hadoop.apache.org/`.

[7] J. Dean and S. Ghemawat, "MapReduce : Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 1–13, 2008.

[8] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proceedings of IPDPS*, pp. 124–124, 2006.

[9] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, Nov. 2007.

[10] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Sharing-aware horizontal partitioning for exploiting correlations during query processing," *Proc. VLDB Endow.*, vol. 3, pp. 542–553, Sept. 2010.

[11] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of ACM SIGMOD*, pp. 517–528, 2012.

[12] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: scaling online social networks," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 375–386, Aug. 2010.

[13] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "MD-HBase: A scalable multi-dimensional data infrastructure for location aware services," 2011.

[14] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[15] F. Reiss, M. N. Garofalakis, and J. M. Hellerstein, "Compact histograms for hierarchical identifiers," in *VLDB*, 2006.

[16] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Fast, small-space algorithms for approximate histogram maintenance," in *STOC*, pp. 389–398, 2002.

[17] A. Deligiannakis, M. N. Garofalakis, and N. Roussopoulos, "Extended wavelets for multiple measures," *ACM Trans. Database Syst.*, vol. 32, no. 2, 2007.

[18] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "One-pass wavelet decompositions of data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 541–554, 2003.

Scalable and Robust Management of Dynamic Graph Data*

Alan G. Labouseur, Paul W. Olsen Jr., and Jeong-Hyon Hwang
{alan, polsen, jhh}@cs.albany.edu
Department of Computer Science, University at Albany – State University of New York, USA

ABSTRACT

Most real-world networks evolve over time. This evolution can be modeled as a series of graphs that represent a network at different points in time. Our G* system enables efficient storage and querying of these *graph snapshots* by taking advantage of the commonalities among them. We are extending G* for highly scalable and robust operation.

This paper shows that the classic challenges of data distribution and replication are imbued with renewed significance given continuously generated graph snapshots. Our data distribution technique adjusts the set of worker servers for storing each graph snapshot in a manner optimized for popular queries. Our data replication approach maintains each snapshot replica on a different number of workers, making available the most efficient replica configurations for different types of queries.

1. INTRODUCTION

Real-world networks, including social networks and the Web, constantly evolve over time [3]. Periodic snapshots of such a network can be represented as graphs where vertices represent entities and edges represent relationships between entities. These *graph snapshots* allow us to analyze the evolution of a network over time by examining variations of certain features, such as the distribution of vertex degrees and clustering coefficients [17], network density [20], the size of each connected component [17, 18], the shortest distance between pairs of vertices [20, 23], and the centrality or eccentricity of vertices [23]. Trends discovered by these analyses play a crucial role in sociopolitical science, marketing, security, transportation, epidemiology, and many other areas. For example, when vertices represent people, credit cards, and consumer goods, and edges represent ownership and purchasing relationships, disruptions in degree distribution, viewed over time, may indicate anomalous behavior, perhaps even fraud.

Several single-graph systems are available today: Google's Pregel [21], Microsoft's Trinity [29], Stanford's GPS [24], the open source Neo4j [22], and others [2, 5, 6, 7, 12, 14]. They, however, lack support for efficiently managing large graph snapshots. Our G* system [13, 27] efficiently stores and queries graph snapshots on multiple worker servers by taking advantage of the commonalities among snapshots. DeltaGraph [16] achieves a similar goal. Our work is complementary to DeltaGraph in that it focuses on new challenges in data distribution and robustness in the context of continuously creating large graph snapshots.

Single-graph systems typically distribute the entirety of a single graph over all workers to maximize the benefits of parallelism. When there are multiple graph snapshots, however, distributing each snapshot on all workers may slow down query execution. In particular, if multiple snapshots are usually queried together, it is more advantageous to *store each snapshot on fewer workers* as long as the overall queried data are balanced over all workers. In this way, the system can *reduce network overhead* (i.e., improve query speed) while *benefiting from high degrees of parallelism*. We present a technique that automatically adjusts the number of workers in a manner optimized for popular queries.

As implied above, there are vast differences in execution time depending on the distribution configurations and the number of snapshots queried together. Replication gives us, in addition to enhanced system reliability, the opportunity to utilize as many distribution configurations as there are replicas. G* constructs r replicas for each snapshot to tolerate up to $r-1$ simultaneous worker failures. Our technique classifies queries into r categories and optimizes the distribution of each replica for one of the query categories.

In this paper, we make the following contributions:

- We define the problem of distributing graph snapshots and present a solution that expedites queries by adjusting the set of workers for storing each snapshot.
- We provide a technique for adaptively determining replica placement to improve system performance and reliability.
- We present preliminary evaluation results that show the effectiveness of the above techniques.
- We discuss our research plans to complete the construction of a highly scalable and reliable system for managing large graph snapshots.

The remainder of the paper is organized as follows: Section 2 presents the research context and provides formal definitions of the problems studied in the paper. Sections 3 and 4 describe our new techniques for distributing and replicating graph snapshots. Section 5 presents our preliminary evaluation results. Section 6 discusses related work. Section 7 concludes this paper.

*This work is supported by NSF CAREER Award IIS-1149372.

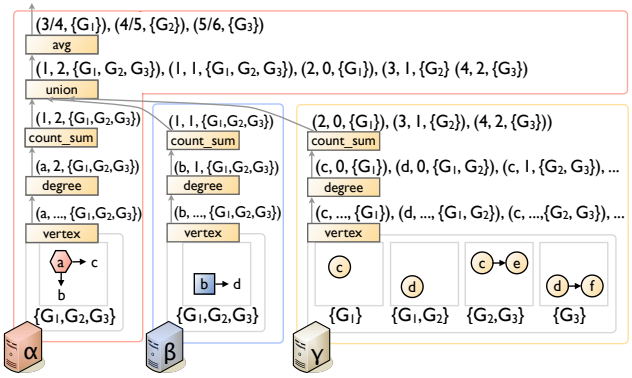


Figure 1: Parallel Calculation of Average Degree

2. BACKGROUND

2.1 Summary of G*

G* is a distributed system for managing large graph snapshots that represent an evolving network at different points in time [13, 27]. As Figure 1 shows, these graph snapshots (e.g., G_1, G_2, and G_3) are distributed over *workers* (e.g., α, β, and γ) that both store and query the graph data assigned to them. The *master* of the system (not shown in Figure 1) transforms each submitted query into a network of operators that process graph data on workers in a parallel fashion. Our previous work on G* can be summarized as follows:

Graph Storage. In G*, each worker efficiently stores its data by taking advantage of commonalities among graph snapshots. Figure 2 shows how worker γ from Figure 1 incrementally stores its portion of snapshots G_1, G_2, and G_3 on disk. The worker stores c_1 and d_1, the first versions of c and d, when it stores G_1. When vertex c obtains a new edge to e in G_2, the worker stores c_2, the second version of c, which shares commonalities with the previous version and also contains a new edge to e. When vertex d obtains a new edge to f in G_3, the worker stores d_2, the second version of d which contains a new edge to f. All of these vertex versions are stored on disk only once regardless of how many graph snapshots they belong to.

To track all of these vertex versions, each worker maintains a *Compact Graph Index* (*CGI*) that maps each combination of vertex ID and graph ID onto the disk location that stores the corresponding vertex version. For each vertex version (e.g., c_2), the CGI stores only one (*vertex ID, disk location*) pair in a collection for the combination of snapshots that contain that vertex version (e.g., $\{G_2, G_3\}$). In this manner, the CGI handles only vertex IDs and disk locations while all of the vertex and edge attributes are stored on disk. Therefore, the CGI can be kept fully or mostly in memory, enabling fast lookups and updates. To prevent the CGI from becoming overburdened by managing too many snapshot combinations, each worker automatically groups snapshots and then separately indexes each group of snapshots [13, 27].

Query Processing. Like traditional database systems, G* supports sophisticated queries using a dataflow approach where operators process data in parallel. To quickly process queries on multiple graph snapshots, however, G* supports special operators that share computations across snapshots.

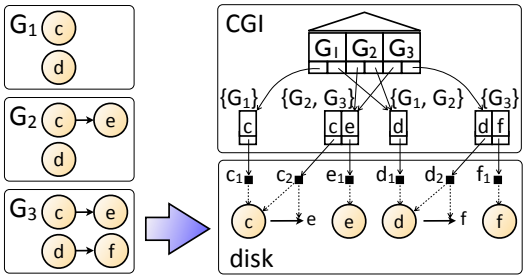


Figure 2: Storage of Snapshots G_1, G_2, G_3 and CGI

PageRank Query	Shared-Nothing	Shared-Everything
One snapshot	285 seconds	22 seconds
All snapshots	285 seconds	2,205 seconds

Table 1: Impact of Graph Snapshot Configuration

Figure 1 shows how the average degree calculation takes place in parallel over three workers. The `vertex` and `degree` operators in Figure 1 compute the degree of each vertex only once while associating the result with all of the relevant graph snapshots (e.g., the degree of vertex a is shared across G_1, G_2, and G_3). In the example, the `count_sum` operators aggregate the degree data, the `union` operator merges these data, and the `avg` operator produces the final result. Details of our graph processing operators and programming primitives for easy implementation of custom operators are provided in our earlier papers [13, 27].

2.2 Problem Statements

Our previous work [13, 27] focused on efficiently storing and querying graph snapshots. We now take up the challenge of doing so in a highly scalable and robust manner.

2.2.1 Multiple Snapshot Distribution

Accelerating computation by distributing data over multiple servers has been a popular approach in parallel databases [10] and distributed systems [9]. Furthermore, techniques for partitioning graphs to facilitate parallel computation have also been developed [15, 24, 25, 26]. However, distributing large graph snapshots over multiple workers raises new challenges. In particular, it is not desirable to use traditional graph partitioning techniques which consider only one graph at a time and incur high overhead given a large number of vertices and edges. Solutions to this problem must (re)distribute *with low overhead* graph snapshots that are continuously generated and take advantage of the property that *query execution time* depends on both *the number of snapshots queried* and the *distribution of the graph snapshots* as illustrated below.

Example. Consider a scenario where each of 100 similarly-sized graph snapshots contains approximately 1 million vertices and 100 million edges. Assume also that the system consists of one master and 100 workers. Table 1 compares two snapshot distribution configurations: *Shared-Nothing*, where each of the 100 snapshots is stored on one distinct worker, and *Shared-Everything*, where each snapshot is evenly distributed over all of the 100 workers. For each of these configurations, two types of queries for computing the PageRank of each vertex are executed: *Query One Snap-*

shot, and *Query All Snapshots*. The explanations below are based on our evaluation results (see Section 5 for details).

In the case of *Shared-Nothing*, querying one snapshot using only one worker takes 285 seconds (205 seconds to construct the snapshot from disk and 80 seconds to run 20 iterations of PageRank). Querying all snapshots on all workers in parallel takes the same amount of time. When the *Shared-Everything* configuration is used, querying one snapshot on all workers takes approximately 22 seconds, mainly due to network communications for the edges that cross worker boundaries (the disk I/O and CPU costs correspond to only 205/100 seconds and 80/100 seconds, respectively, due to the distribution of the snapshot over 100 workers). In this configuration, querying 100 snapshots takes 2,205 seconds as the PageRank of each vertex varies across graph snapshots, thereby causing 100 times more message transmissions than the previous case. This example shows the benefits of different snapshot distribution approaches for different types of queries (e.g., *Shared-Nothing* for *queries on all snapshots* and *Shared-Everything* for *queries on one snapshot*).

Formal Definition. Our ultimate goal is to keep track of the popularity of graph snapshots and to optimize the storage/distribution of *unpopular* snapshots for space efficiency (Section 2.1) and *popular snapshots* for query speed. In this paper, we focus on the problem of distributing popular snapshots over workers in a manner that minimizes the execution time of queries on these snapshots. This problem can be formally defined as follows:

PROBLEM 1. **(Snapshot Distribution)** *Given a series of graph snapshots* $\{G_i(V_i, E_i) : i = 1, 2, \cdots\}$*, n workers, and a set of queries Q on some or all of the snapshots, find a distribution* $\{V_{i,w} : i = 1, 2, \cdots \wedge w = 1, 2, \cdots, n\}$ *that minimizes* $\sum_{q \in Q} time(q, \{V_{i,w}\})$ *where* $V_{i,w}$ *denotes the set of vertices that are from snapshot* $G_i(V_i, E_i)$ *and that are assigned to worker w, and* $time(q, \{V_{i,w}\})$ *represents the execution time of query* $q \in Q$ *on the distributed snapshots* $\{V_{i,w}\}$ *satisfying (1)* $\cup_{w=1}^{n} V_{i,w} = V_i$ *(i.e., the parts of a snapshot on all workers cover the original snapshot) and (2)* $V_{i,w} \cap V_{i,w'} = \emptyset$ *if* $w \neq w'$ *(i.e., workers are assigned disjoint parts of a snapshot).*

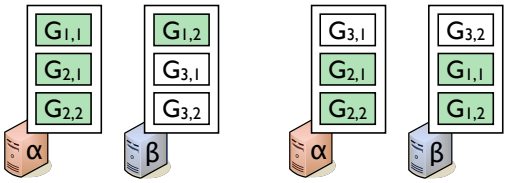
Our solution to the above problem is presented in Section 3.

2.2.2 Snapshot Replication

There have been various techniques for replicating data to improve availability and access speed [8, 11, 28]. A central data replication challenge in G* is to distribute each replica of a snapshot over a possibly different number of workers to maximize both performance and availability. For each query, the most beneficial replica also needs to be found according to the characteristics of the query (e.g., the number of snapshots queried). If two replicas of a graph snapshot are distributed using the *Shared-Nothing* and *Shared-Everything* approaches, queries on a single snapshot should use the *Shared-Everything* replica configuration rather than the other. In practice, however, each query can access an arbitrary number of graph snapshots (not necessarily one or all), thereby complicating the above challenges. The problem of replicating graph snapshots can be defined as follows:

PROBLEM 2. **(Snapshot Replication)** *Given a series of graph snapshots* $\{G_i(V_i, E_i) : i = 1, 2, \cdots\}$*, the degree of replication r, n workers, and a set of queries Q on some or all of the snapshots, find a replica distribution* $\{V_{i,j,w} : i = 1, 2, \cdots \wedge j = 1, 2, \cdots, r \wedge w = 1, 2, \cdots, n\}$ *that minimizes* $\sum_{q \in Q} time(q, \{V_{i,j,w}\})$ *where* $V_{i,j,w}$ *denotes the set of vertices that are from the jth replica* $G_{i,j}(V_{i,j}, E_{i,j})$ *of snapshot* $G_i(V_i, E_i)$ *and that are assigned to worker w, and* $time(q, \{V_{i,j,w}\})$ *denotes the execution time of query q on the distributed snapshot replicas* $\{V_{i,j,w}\}$ *satisfying (1)* $\cup_{w=1}^{n} V_{i,j,w} = V_{i,j} = V_i$ *for* $j = 1, 2, \cdots, r$*, (i.e., the parts of a snapshot replica on all workers cover the original replica), (2)* $V_{i,j,w} \cap V_{i,j,w'} = \emptyset$ *if* $w \neq w'$ *(i.e., workers are assigned disjoint parts of a snapshot replica), and (3)* $V_{i,j,w} \cap V_{i,j',w} = \emptyset$ *if* $j \neq j'$ *(i.e., no worker w contains multiple copies of a vertex and its edges, which tolerates* $r - 1$ *simultaneous worker failures).*

Section 4 presents our solution to the above problem.



(a) Before Exchange (b) After Exchange

Figure 3: Exchanging Segments. If snapshots G_1 and G_2 are queried together frequently, workers α and β in Figure 3(a) can better balance the workload and reduce the network overhead by swapping $G_{1,1}$ and $G_{3,1}$.

3. GRAPH SNAPSHOT DISTRIBUTION

As mentioned in Section 2.2.1, G* needs to store each graph snapshot on an appropriate number of workers while balancing the utilization of network and CPU resources. In contrast to traditional methods for partitioning a *static* graph [15, 25], G* must determine the location of each vertex and its edges on the fly in response to a continuous influx of data from external sources.

Our *dynamic* data distribution approach meets the above requirements. In this approach, each G* worker partitions its graph data into *segments* with a certain maximum size (e.g., 10GB) so that it can control its load by migrating some segments to other workers (Section 3.1). Our approach continuously routes incoming messages for updating vertices and edges to appropriate workers with low latency (Section 3.2). When a segment becomes full, G* splits that segment into two that are similar in size while maintaining data locality by keeping data accessed together within the same segment (Section 3.3). It does all of the above while supporting G*'s graph processing operators (Section 3.4).

3.1 Load Balancing

In G*, each worker periodically communicates with a randomly chosen worker to balance graph data. Our key principles in load balancing are to (1) *maximize the benefits of parallelism by uniformly distributing data that are queried together* and (2) *minimize network overhead by co-locating data from the same snapshot*. Consider Figure 3(a) where three snapshots are partitioned into a total of 6 similarly-sized segments. In this example, each of workers α and β are assigned a segment from snapshot G_1, α is assigned two segments from G_2, and β is assigned two segments from G_3. If snapshots G_1 and G_2 are frequently queried together

(see those shaded in Figure 3(a)), this snapshot distribution leads to inefficient query execution due to imbalanced workload between the workers and network communications for the edges between $G_{1,1}$ and $G_{1,2}$. This problem can be remedied by exchanging $G_{1,1}$ and $G_{3,1}$ between the workers, which results in a balanced distribution of the data queried together (i.e., G_1 and G_2) and localized processing of G_1 on β and G_2 on α, respectively.

Given a pair of workers, our technique estimates, for each segment, the benefit of migrating that segment to the other worker, and then performs the most beneficial migration. This process is repeated a maximum number of times or until the migration benefit falls below a predefined threshold. The benefit of migrating a segment is calculated by multiplying the probability that the segment is queried with the expected reduction in query time (i.e., the difference between expected query time before and after migration).

For a set S_i of segments on worker i and another set S_j of segments on worker j, the expected query time is computed as $\sum_{q \in \mathcal{Q}_k} p(q) \cdot time(q, S_i, S_j)$ where $\mathcal{Q}_k$ is a collection of k popular query patterns, $p(q)$ is the probability that query pattern q is executed, and $time(q, S_i, S_j)$ denotes the estimated duration of q given segment placements S_i and S_j.

Our technique obtains $\mathcal{Q}_k$ (equivalently, k popular combinations of segments queried together) as follows: Sort segments from $S_i \cup S_j$ in order of decreasing popularity. Initialize $\mathcal{Q}_k$ (for storing k popular query patterns) with the first segment. Then, for each of the remaining segments, combine it with each element from $\mathcal{Q}_k$ and insert the result back into $\mathcal{Q}_k$. Whenever $|\mathcal{Q}_k| > k$, remove its least popular element. We estimate the popularity of each combination of segments by consolidating the counting synopses [4] for those segments. Whenever a query accesses a segment, the associated synopsis is updated using the ID of the query.

We compute $time(q, S_i, S_j)$ as $\max(c(q, S_i), c(q, S_j)) + c'(q, S_i, S_j)$ where $c(q, S_i)$ is the estimated duration of processing the segments from S_i for query q, and $c'(q, S_i, S_j)$ represents the estimated time for exchanging messages between workers i and j for query q.

3.2 Updates of Vertices and Edges

Each new vertex (or any edge that emanates from the vertex) is first routed to a worker chosen according to the hash value of the vertex ID. That worker assigns such a vertex to one of its data segments while saving the (*vertex ID*, *segment ID*) pair in an index similar to the CGI (Section 2.1). If a worker receives an edge that emanates from an existing vertex v, it assigns that edge to the segment that contains v. If a worker w has created a segment S and then migrated it to another worker w' for load balancing reasons (Section 3.1), worker w forwards the data bound to S to w'. To support such data forwarding, each worker keeps track of the worker location of each data segment that it has created before. Updates of vertices and edges, including changes in their attribute values, are handled as in the case of edge additions. This assignment of graph data to workers is scalable because it distributes the overhead of managing data over workers. It also proceeds in a parallel, pipelined fashion without any blocking operations.

3.3 Splitting a Full Segment

If the size of a data segment reaches the maximum (e.g., 10GB), the worker that manages the segment creates a new segment and then moves a half of the data from the previous segment to the new segment. To minimize the number of edges that cross segment boundaries, we use a traditional graph partitioning method [15]. Whenever a segment is split as above, the worker also updates the (*vertex ID, segment ID*) pairs for all of the vertices migrated to the new segment. This update process incurs relatively low overhead since the index can usually be kept in memory as in the case of the CGI (Section 2.1). If a worker splits a segment which was obtained from another worker, it sends the update information to the worker that originally created it in order to enable data forwarding as mentioned in Section 3.2.

3.4 Supporting Graph Processing Operators

G*'s graph processing operators, such as those for computing clustering coefficients, PageRank, or the shortest distance between vertices, are usually instantiated on every worker that stores relevant graph data [13]. These operators may exchange messages to compute a value for each vertex (e.g., the current shortest distance from a source vertex). If an operator needs to send a message to a vertex, the message is first sent to the worker whose ID corresponds to the hash value of the vertex ID. This worker then forwards the message to the worker that currently stores the vertex. This forwarding mechanism is similar to that for handing updates of vertices and edges (Section 3.2).

4. GRAPH SNAPSHOT REPLICATION

G* masks up to $r-1$ simultaneous worker failures by creating r copies of each graph data segment. As discussed in Sections 2.2 and 3, the optimal distribution of each graph snapshot over workers may vary with the number of snapshots frequently queried together. Based on this observation, we developed a new data replication technique that speeds up queries by configuring the storage of replicas to benefit different categories of queries. This approach uses an online clustering algorithm [1] to classify queries into r categories based on the number of graphs that they access. It then assigns the j-th replica of each data segment to a worker in a manner optimized for the j-th query category. The master and workers support this approach as follows:

4.1 Updates of Vertices and Edges

Updates of vertices and edges are handled as described in Section 3.2 except that they are routed to r data segment replicas on different workers. For this reason, each worker keeps a mapping that associates each segment ID with the r workers that store a replica of the segment. Our approach protects this mapping on worker w by replicating it on workers $(w+1)\%n, (w+2)\%n, \cdots, (w+r-1)\%n$ where n denotes the number of workers. If a worker fails, the master assigns another worker to take over.

4.2 Splitting a Full Segment

The replicas of a data segment are split in the same way due to the use of a deterministic partition method. For each vertex migrated from one data segment to another, the r workers that keep track of that vertex update their (*vertex ID, segment ID*) pairs accordingly.

4.3 Query-Aware Replica Selection

For each query, the master identifies the worker locations of the data segment replicas to process. To this end, the

Message passing (12-bytes/message)	1M messages/sec
Disk I/O bandwidth	200 Mbytes/sec
Snapshot construction in memory	200 seconds
PageRank iteration per snapshot	4 seconds

Table 2: Speed and Bandwidth Observations

# Cores	1	2	4	8	16	24	48
Speedup	1.0	1.9	3.7	5.9	9.7	12.5	14.7

Table 3: Actual Speedup Result

master keeps track of the mapping between graph snapshots and the data segments that constitute them. The master also maintains the mapping between data segment replicas and the workers that store them. Using these mappings, the master selects one replica for each data segment such that the overall processing load is uniformly distributed over a large number of workers and the expected network overhead is low. Next, as Figure 1 shows, the master instantiates operators on these workers and starts executing the query.

4.4 Load Balancing

Each worker balances its graph data as explained in Section 3. The only difference is that whenever a query of category j accesses a replica of a data segment, the counting synopsis of the j-th replica of the data segment is updated using the ID of the query (Section 3.1). In this way, the j-th replica of each segment is assigned to a worker in a manner optimized for query category j.

5. PRELIMINARY EVALUATION

This section presents our preliminary results obtained by running G* on a six-node, 48-core cluster. In this cluster, each machine has two Quad-Core Xeon E5430 2.67 GHz CPUs, 16GB RAM, and a 2TB hard drive. We plan to extend these experiments with more queries on larger data sets in a bigger cluster (Section 7).

To construct a realistic example in Section 2.2.1, we measured the overhead of key operations summarized in Table 2. In our evaluation, a worker was able to transmit up to 1 million messages to other workers within a second, although a 1Gbps connection may enable 10 million transmissions of 12-byte messages in theory. The reason behind this result is that there is inherent overhead when writing and creating message objects to and from TCP sockets in Java. Furthermore, reading approximately 1Gbytes of data from disk to construct a graph snapshot took 5 seconds. However, constructing a snapshot in memory by creating 100 million edge objects and registering them in an internal data structure took approximately 200 seconds.

In the next set of experiments, we created a series of 500 graph snapshots using a binary tree generator. Each snapshot in the series was constructed by first cloning the previous snapshot and then inserting 20,000 additional vertices and edges to the new graph. Therefore, the last graph in the series contained 10 million vertices. We ran a query that computes, for each graph, the distribution of the shortest distances from the root to all other vertices. Table 3 shows, for the shortest distance query, the speedup achieved by distributing the snapshots over more workers. The highest speedup was achieved with 48 workers. This table also shows that the relative benefit of data distribution (i.e., the speedup relative to the number of workers) tends to decrease with more workers. This is mainly due to increased network traffic, which shows the importance of balancing CPU and network resources in the context of continuously creating large graph snapshots.

SSSP Query	All Workers	Subset of Workers
One snapshot	8.2 seconds	19.2 seconds
All snapshots	80.5 seconds	53.2 seconds

Table 4: Impact of Graph Data Distribution

The effectiveness of two different distributions is demonstrated in Table 4. If most queries access only the largest snapshot, then it is beneficial to distribute that snapshot over all workers to maximize query speed. On the other hand, if all of the snapshots are queried together, our approach stores each graph on a smaller subset of workers to reduce network overhead. In this case, all of the workers can still be used in parallel since the entire graph data is distributed over all workers. The benefits of distribution configurations are less pronounced in Table 4 than Table 1 due to a smaller number of message transmissions and fewer workers. Table 4 also demonstrates the benefit of G* in executing queries on multiple snapshots. In particular, the time for processing 500 snapshots (e.g., 80.5 seconds) is only up to 10 times longer than that for processing the largest snapshot (e.g., 8.2 seconds) since the computations on the largest snapshot are shared across smaller snapshots.

6. RELATED WORK

In this section, we briefly summarize related research, focusing on previous graph systems, data distribution, and data replication.

Previous Graph Systems. In contrast to systems which process one graph at a time [2, 5, 6, 7, 12, 14, 21, 22, 24, 29], G* efficiently executes sophisticated queries on multiple graph snapshots. G*'s benefits over previous systems are experimentally demonstrated in our prior work [13]. DeltaGraph [16] and GraphChi [19] are promising systems for dynamic graphs but do not directly address the data distribution/replication issues considered in this paper.

Data Distribution. Traditional graph partitioning techniques split a static graph into subgraphs in a manner that minimizes the number of crossing edges [15, 25]. There are also recent graph repartitioning schemes that observe communication patterns and then move vertices to reduce network overhead [24, 26]. In contrast to them, our technique dynamically adjusts the number of workers that store each graph snapshot according to the real-time influx of graph data and popular types of queries (Section 3).

Data Replication. There has been extensive work on data replication that focused on improving data availability and performance [8, 11, 28]. Researchers developed techniques for ensuring replica consistency [11] and finding most advantageous replica placement [8]. Stonebraker et al. proposed an approach that stores each database replica differently, optimized for a different query type [28]. While our replication approach has some similarity in terms of high-level ideas, it is substantially different in that it distributes each

graph snapshot over a different number of workers to speed up different types of queries.

7. CONCLUSIONS AND FUTURE WORK

We presented G*, a scalable and robust system for storing and querying large graph snapshots. G* tackles new data distribution and replication challenges that arise in the context of continuously creating large graph snapshots. Our data distribution technique efficiently stores graph data on the fly using multiple worker servers in parallel. This technique also gradually adjusts the number of workers that store each graph snapshot while balancing network and CPU overhead to maximize overall performance. Our data replication technique maintains each graph replica on a different number of workers, making available the most efficient storage configurations for various combinations of queries.

We are working on full implementations of the techniques presented in this paper to enable new experiments with additional queries on larger data sets. We will analyze these techniques to classify their complexity. We plan to look into the challenges of scheduling groups of queries, dealing with varying degrees of parallelism, resource utilization, and user-generated performance preferences. We are exploring failure recovery techniques for long-running queries while exposing the tradeoff between recovery speed and execution time. We also want to study opportunities for more granular splitting, merging, and exchanging of data at the vertex and edge level rather than in large segments as discussed in this paper. We intend to seek opportunities for gains in execution speed at the expense of storage space by segregating recent and popular "hot" data (which we could store in a less compressed manner) from less popular "cold" data (which could be highly compressed).

8. REFERENCES

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A Framework for Clustering Evolving Data Streams. In *VLDB*, pages 81–92, 2003.

[2] Apache Hama. *http://hama.apache.org*.

[3] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. PageRank on an Evolving Graph. In *KDD*, pages 24–32, 2012.

[4] K. S. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On Synopses for Distinct-Value Estimation Under Multiset Operations. In *SIGMOD*, pages 199–210, 2007.

[5] Cassovary. Open Sourced from Twitter *https://github.com/twitter/cassovary*.

[6] A. Chan, F. K. H. A. Dehne, and R. Taylor. CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *IJHPCA*, 19(1):81–97, 2005.

[7] R. Chen, X. Weng, B. He, and M. Yang. Large Graph Processing in the Cloud. In *SIGMOD*, pages 1123–1126, 2010.

[8] Y. Chen, R. H. Katz, and J. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *IPTPS*, pages 306–318, 2002.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[10] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. Gamma - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[11] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, pages 173–182, 1996.

[12] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *POOSC*, 2005.

[13] J.-H. Hwang, J. Birnbaum, A. Labouseur, P. W. Olsen Jr., S. R. Spillane, J. Vijayan, and W.-S. Han. G*: A System for Efficiently Managing Large Graphs. Technical Report SUNYA-CS-12-04, CS Department, University at Albany – SUNY, 2012.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In *ICDM*, pages 229–238, 2009.

[15] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *SC*, page 29, 1995.

[16] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. *CoRR*, abs/1207.5777, 2012.

[17] G. Kossinets and D. J. Watts. Empirical Analysis of an Evolving Social Network. *Science*, 311(5757):88–90, 2006.

[18] R. Kumar, J. Novak, and A. Tomkins. Structure and Evolution of Online Social Networks. In *KDD*, pages 611–617, 2006.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[20] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking diameters and Possible Explanations. In *KDD*, pages 177–187, 2005.

[21] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[22] Neo4j The Graph Database. *http://neo4j.org/*.

[23] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. *PVLDB*, 4(11):726–737, 2011.

[24] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[25] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. Technical Report TR 00-018, Computer Science and Engineering, U. of Minnesota, 2000.

[26] Z. Shang and J. X. Yu. Catch the Wind: Graph Workload Balancing on Cloud. In *ICDE*, pages 553–564, 2013.

[27] S. R. Spillane, J. Birnbaum, D. Bokser, D. Kemp, A. Labouseur, P. W. Olsen Jr., J. Vijayan, and J.-H. Hwang. A Demonstration of the G* Graph Database System. In *ICDE*, pages 1356–1359, 2013.

[28] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[29] Trinity. *http://research.microsoft.com/en-us/projects/trinity/*.

Towards Elastic Stream Processing: Patterns and Infrastructure

Kai-Uwe Sattler
Ilmenau University of Technology
Ilmenau, Germany
kus@tu-ilmenau.de

Felix Beier*
Ilmenau University of Technology
Ilmenau, Germany
felix.beier@tu-ilmenau.de

ABSTRACT

Distributed, highly-parallel processing frameworks as Hadoop are deemed to be state-of-the-art for handling big data today. But they burden application developers with the task to manually implement program logic using low-level batch processing APIs. Thus, a movement can be observed that high-level languages are developed which allow to declaratively model dataflows that are automatically optimized and mapped to the batch-processing backends. However, most of these systems are based on programming models as MapReduce that provide elasticity and fault-tolerance in a natural manner since intermediate results are materialized and, therefore, processes can simply be restarted and scaled with partitioning input datasets. For continuous query processing on data streams, these concepts cannot be applied directly since it must be guaranteed that no data is lost when nodes fail. Usually, these long running queries contain operators that maintain state information which depends on the data that has already been processed and hence they cannot be restarted without information loss. This also is an issue when streaming tasks should be scaled. Therefore, integrating elasticity and fault-tolerance in this context is a challenging task which is subject of this paper. We show how common patterns from parallel and distributed algorithms can be applied to tackle these problems and how they are mapped to the Mesos cluster management system.

1. INTRODUCTION

Processing and analyzing big data is one of todays big challenges. A popular definition from a Gartner report names the *three 'V's – volume*, *velocity*, and *variety* as the main characteristics of big data. Among them, velocity refers to the analytics of dynamic data even in (near) realtime.

Several approaches and techniques have been developed in the past to process dynamic data. *Data stream management systems (DSMS)* like STREAM, Aurora, IBM Infosphere Streams or our own AnduIN engine provide abstractions to process continuous and possibly infinite streams of data instead of disk-resident datasets. Typically, this includes standard (relational) query operators, window-based operators for computing joins and aggregations as well as more advanced data analytics and data mining operators working on portions of the stream, e.g. windows or synopses of data. *Complex Event Processing systems (CEP)* particularly support the identification of event patterns in (temporal) streams of data such as a sequence of specific event types within a given time interval. Typically, systems of both classes provide a declarative interface, either in form of SQL-like query languages like CQL for DSMS, event languages like SASE, or in the form of dataflow specifications like SPL in IBM Infosphere Streams.

Recently, several new distributed stream computing platforms have been developed, aiming at providing scalable and fault-tolerant operation in cluster environments. Examples are Apache S4 or Storm. In contrast to DSMS or CEP engines theses platforms do not (yet) provide declarative interfaces and, therefore, require to program applications instead of writing queries. Developers of these systems argue that they provide the same for stream processing what Hadoop did for batch processing – which raises the hope of a similar movement towards higher-level languages as we can see with Pig, Jaql etc. for MapReduce.

However, there are some challenges in scalable and elastic stream processing which are different from batch processing with Hadoop. Whereas in Hadoop, input data as well as intermediate results are materialized on disk and, therefore,

- both, map and reduce tasks can be restarted arbitrarily in case of failures until the entire job is finished,
- since computation state is saved, the number of nodes assigned to map and reduce tasks can be simply adjusted by partitioning input and intermediate results.

This is more difficult when processing dynamic data – even with platforms as S4 or Storm which to some extent support a reliable and scalable operation. The main differences are:

(1) Partitioning of streams for data-parallel processing is not always easily possible, for example in case of window- or sequence-based operators including CEP operators. Also, elastic operation by adding new nodes at runtime of a query requires at least rerouting of data.

(2) Stream queries are typically long running queries which cannot be simply restarted without losing data. Furthermore, because of this, the deployment and resource allocation (placement of queries on nodes, allocating memory and CPUs) are much more critical.

*This work is partially funded by an IBM PhD Fellowship.

In this paper, we try to answer the question how to bridge the gap between an easy-to-use, high-level declarative interface for data stream analytics and scalable cluster-based stream computing platforms in order to address these challenges. The contribution of this paper is twofold:

- Based on a basic dataflow model for stream queries we describe patterns for fault-tolerant and scalable query processing and discuss constraints of their application.
- We show the implementation and deployment of these patterns using our distributed stream and CEP engine AnduIN and the Mesos cluster infrastructure by describing techniques supporting flexible and elastic deployment.

Though, we use the AnduIN system for describing and implementing the concepts, we think the ideas and patterns are applicable to other platforms, too.

2. RELATED WORK

The relevant work related to this paper can be classified into the main categories: continuous query processing and scalable dataflow platforms.

Continuous query processing is usually implemented in data stream management systems (DSMS). Pioneered by systems like STREAM, Borealis, and Telegraph, several approaches and systems have been developed in the last decade including commercial products such as IBM InfoSphere Streams and StreamBase. Typically, these systems provide a SQL-like query language enhanced by features for dealing with continuous queries such as sliding windows.

Partitioning, distributed processing, and fault tolerance have been studied to some extent, e.g., in Borealis [1] by introducing replicated processing nodes as well as several new tuple types such as punctuation tuples and control tuples like undo tentative (tuples resulting from processing a subset of the input which can be corrected later) and done tuples indicating that state reconciliation finished. State reconciliation is the process of stabilizing the output result, e.g., by replacing previously tentative results. In this way, this approach aims at fault-tolerance but not at partitioning.

Another approach in the form of a programming model has been proposed in [14] as so-called discretized streams (D-Streams). This idea is based on resilient distributed datasets which are storage abstractions used for rebuilding lost data.

An approach addressing load balancing issues by partitioning while providing fault tolerance for pipelined dataflows is Telegraph's FluX [10]. FluX is a dataflow operator extending the idea of the exchange operator form parallel query processing. The operator encapsulates state partitioning and tuple routing and allows to repartition even stateful operators while executing the dataflow pipeline.

Scalable dataflow platforms try to extend the applicability of the MapReduce paradigm for large-scale parallel batch processing to pipeline processing and continuous query support. One example is HOP (Hadoop Online Prototype) [4]. In HOP, map tasks maintain TCP sockets to reducers for pipelining their output. In addition, pipelining is also supported between jobs by sending the output of reducers directly to mappers of a subsequent job. Further, distributed dataflow systems are Twitter's Storm and Apache S4. Storm implements fault detection at task level and guaranteed message passing, whereas in S4 messages can be lost. Storm runs so-called topologies – subsets of these topologies are assigned to worker processes of a cluster. However, these systems only offer a simple programming model and, therefore, operators and topologies have to be implemented in a programming language like Java or Python. Furthermore, state recovery and partitioning have to be implemented manually, too.

Optimus [11] is a framework for dynamic rewriting of execution plans for data-parallel computing, e.g., formulated in DryadLINQ. The framework supports rewriting of MapReduce programs at runtime, addressing issues like repartitioning, fault tolerance, and handling data skew. But the required algorithms have to be implemented by the user.

3. PATTERNS FOR SCALABLE PROCESSING OF DATA STREAM QUERIES

In the following we assume a simple processing model for data stream queries: a query is represented by a dataflow graph which is a common model in literature [9, 11].

In such a directed acyclic graph, nodes represent query operators and edges describe the tuple flow between them. Query nodes can be arbitrary pipeline operators of a stream query algebra [2] like filter, projection etc. as well as window- or synopsis-based operators as sliding window joins and aggregations, but also more complex data analytics operators including CEP and data mining. Communication between query operators is performed either directly by invoking operator functions or via buffers/queues. Obviously, this represents a very generic execution model to which a wide range of declarative query languages like CQL, dataflow specifications such as IBM's SPL, or implementation-oriented approaches as used in S4 or Storm can be mapped. This model can be easily extended to the distributed case by inserting network reader/writer nodes which use appropriate communication protocols and APIs, e.g., TCP/UDP sockets or more advanced solutions like ZeroMQ.

There are two main reasons for distributing query nodes: increasing processing reliability by introducing redundancy, and increasing performance and/or scalability by load distribution. The following patterns support these goals to different degrees. In this discussion, we use the term "query task" as the unit of distribution/scheduling for both, elementary algebra operators, and for dataflow (sub-)graphs with well-defined properties (input/output, stateless vs. statefulness).

Pattern 1: Simple standby: For a critical query node N a standby node S is maintained on a separate compute unit which is activated if N fails. This requires monitoring of N, e.g. by a combination of heartbeats and cluster coordination service such as ZooKeeper as well as rerouting the input tuple stream to S. Since in case of a failure the state of N is lost, this pattern is applicable only for stateless nodes.

Pattern 2: Checkpointing: This pattern is similar to pattern 1 but supports stateful operators. Failover is achieved by periodically checkpointing the state of the critical node N to a shared disk and restarting the standby node S from the checkpoint. Examples of such checkpoints are the content of sliding windows or hash tables for joins and aggregations.

Pattern 3: Hot standby: If the failover time of pattern 2 is not acceptable, a hot standby approach can be chosen where redundant query nodes S are kept actively. To achieve this, the input stream has to be sent to all redundant nodes, either by using multicast strategies at the network or at the application level. This pattern works both with stateless and stateful query operators but requires a special node to eliminate duplicate results, e.g., a stream selector node which forwards the input from only one of multiple streams.

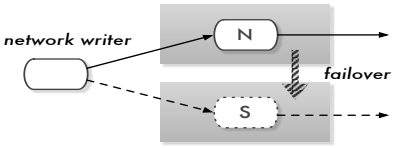


Figure 1: Simple Standby

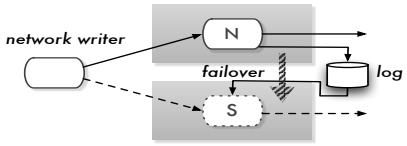


Figure 2: Checkpointing

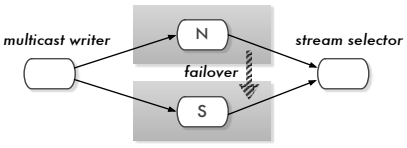


Figure 3: Hot Standby

Pattern 4: Stream partitioning: This pattern exploits data parallelism by partitioning the input stream. It can be implemented by a splitter node redirecting each input tuple to one of the query nodes $N_1 \dots N_k$ or by a multicast writer with an additional partitioning node $P_1 \dots P_k$ for filtering the input stream according to the partitioning scheme. Finally, the results are merged into a single stream.

Pattern 5: Stream pipelining: In contrast to pattern 4 this pattern exploits task parallelism by splitting a complex query node N into a sequence of query nodes $N_1 \dots N_k$ and placing them on separate compute units.

Usually, multiple patterns will be applied in order to achieve certain quality of service (QoS) guarantees as fault tolerance (patterns 1-3) or elasticity for adapting resource consumption (patterns 4-5) according to the needs of the applications. Of course this pattern list does not claim completeness. There are several others that are applicable under certain circumstances, e.g., parallelization through aggregation trees for commutative and associative aggregation operators [9]. Nevertheless, these basic patterns described here are well-known in distributed and parallel algorithms, and – with slight modifications – cover various use cases [1].

In the following we will describe how these patterns can be utilized in a dataflow framework to dynamically restructure the physical representation of the graph in a continuous query context that is executed in a cluster infrastructure. The restructuring is achieved with a simple set of rewriting rules that are automatically applied on the graph without the need to manually code them as in existing approaches [11] while guaranteeing that no state information is lost during the restructuring phase. We present the algorithms based on our AnduINv2 stream processing engine but highlight that they are also applicable on other frameworks as Dryad with slight modifications to achieve streaming semantics.

4. QUERY DEPLOYMENT INFRASTRUCTURE

Fig. 6 illustrates the dataflow model used in AnduINv2 and how it is mapped to the physical layer of executable code. While the first prototype of the system aimed at processing sensor data as well as in-network processing [12] and complex event processing (CEP) [8], our current research focuses on processing techniques for cluster environments. The AnduINv2 system comprises three components: (1) the runtime environment containing the implementation of query operators including a CEP engine as well as operators for controlling the query execution, (2) a query compiler translating a dataflow-based query specification given in an XML file into query tasks, i.e., executable code linked to the runtime environment, (3) a query scheduler and executor integrated with the Mesos cluster framework that deploys query tasks for processing them on physical nodes.

[1]Actually, aggregation trees are just combinations of partitioning and pipelining patterns.

AnduINv2 queries are deployed as separate processes in Mesos which are just-in-time compiled using the system's C++ compiler. This provides an easy mechanism to plug-in user defined operators and exchange operator implementations. During deployment, processes are interlinked by query channels which simply represent an abstraction over network connections (TCP/UDP sockets, ZeroMQ connections).

Query tasks can be shared among multiple queries when they share some common (sub)streams or operators as described in [3]. Further, a query can be implemented as a set of tasks which are distributed across multiple nodes in the cluster. Therefore, the logical query tree is partitioned into smaller subtrees that are translated separately. We will use these mechanisms for implementing the elasticity patterns.

4.1 Dataflow Graph Rewriting

dataflows are specified in an XML format which can be seen as an intermediate representation, allowing to use different frontends such as CQL or graphical tools. A dataflow specification consists of stream type definitions and operator definitions with name, type, type-specific parameters as well as input and output channels. These channels are typed and are used to interconnect operators to form a graph. The following example shows a simple dataflow specification. (We omitted the XML notation for better readability).

```
type name = "aStreamType" {
    column name = "x", type = "int" ...
}
operator name = "source", type = "reader" {
    output name = "aStream", type ="aStreamType"
}
operator name = "myFilter", type = "filter" {
    input name = "aStream"
    param condition = "aStream.x < 42"
    output name = "filteredStream", type = "aStreamType"
}
operator name = "sink", type = "writer" {
    input name = "filteredStream"
}
```

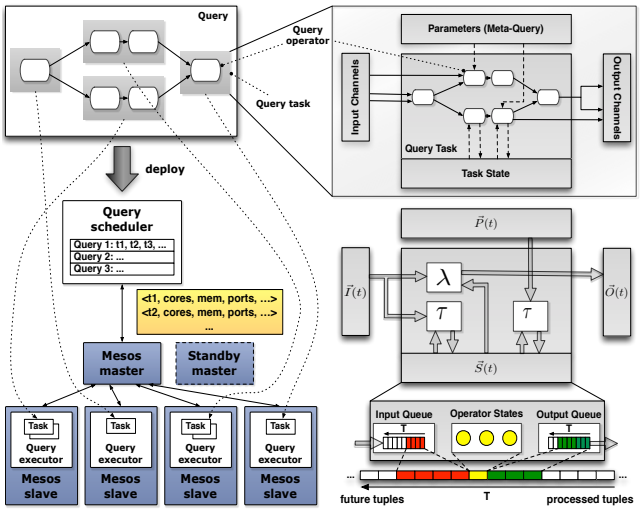


Figure 6: query model

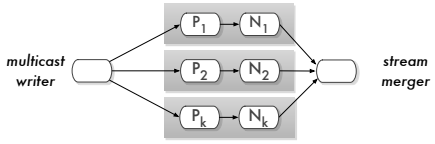


Figure 4: Partitioning

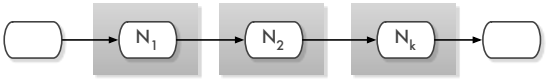


Figure 5: Pipelining

Note, that apart from stream input and possible output no communication operators have to be specified as part of the query. Such operators are added during rewriting if necessary. For formulating rewriting rules we use a simple notation. A dataflow as given above is written as

$$\mathsf{sink := writer(f := filter(src := reader))}$$

where writer, filter, and reader are operator types and the optional sink, f, and src names denote operator instances.

During rewriting, graph patterns have to be matched and constraints are checked. For this purpose, the pseudo-type any is used as a placeholder for any possible type, and any* represents a dataflow of arbitrary operator types. The following pattern matches a dataflow subgraph containing a stateless filter operator (which is the case for any filter):

$$\mathsf{a_2 := any(f := filter(a_1 := any^*))[stateless(f)]}$$

To apply the patterns, a rewriting rule can be specified:

$$\Rightarrow \underbrace{\mathsf{a_2(stream\text{-}selector(failover(}}_{@p_1}\underbrace{\mathsf{writer(f(}}_{@p_*}\underbrace{\mathsf{reader(multicast(a_1)))))))}}_{@p_2}$$

Besides inserting or replacing operators (such as stream-selector, failover, and multicast operators in the previous example), operator nodes are also annotated with placement information where p_i, p_j with $i \neq j$ denote distinct compute nodes and p_* denotes an arbitrary number of nodes.

4.2 Failover Handling

With these rewriting rules, internal data management nodes and different operator implementations can be transparently injected into the query plan without impacting the application. This allows to re-schedule a query task to another node in case of a failure (pattern 1), use an operator implementation that automatically integrates snapshotting (pattern 2), or replicate a task to implement hot standby.

The actual flow of data through query channels during runtime is controlled by special operator parameters – e.g., target IP addresses and ports – that can be adjusted through a concept we call *meta queries* (cf. Sect. 4.5). To detect and react on failures, query tasks are instrumented with monitoring interfaces that inform the query scheduler about the nodes' health and performance measures as tuple processing rates. The scheduler then triggers a graph rewriting.

When a rewritten graph needs to be deployed, it has to be guaranteed that no information of the tuple stream is lost. To analyze the necessary steps, we reduce a query task to a finite state machine model (cf. Fig. 6) which is common for implementing CEP operators [6] but can also be applied for general dataflow transformations. The query task receives a stream of input tuples $\vec{I}(t)$, applies its logical operation(s) λ – e.g., a filter or a join – to generate an output stream $\vec{O}(t)$. (We use vectorial representations here to combine all channels into a single quantity.) The output might depend on the task's state $\vec{S}(t)$, i.e., the state of all internal operators which can basically include anything that is required for implementing the operators (e.g., hash tables, or sliding windows) and are updated with each incoming tuple through a state transition function τ_i [2]. The meta query extension is represented by special input channels $\vec{P}(t)$ that modify the operator state through τ_p.

To guarantee that a node failure does not lead to an information loss it is necessary that all results which have not been consumed by the following target can be reproduced from the possibly infinite input stream. Therefore, the operator state needs to be snapshotted after each input tuple, or – if this is too expensive – the input tuples need to be persisted in order to reproduce this state with just 'replaying' the input. Which tuples are still required for a possible replay can be controlled by special tuple messages that are exchanged between tuple producers and consumers as in the Borealis system [1]. Note when frameworks as ZeroMQ are used to implement query channels, reliable message delivery can be guaranteed without the need to modify operators.

In order to implement fault tolerance with transferring stateful query tasks to other computing nodes, a simple protocol as presented in [10] is sufficient:
(1) quiesce all input streams,
(2) replicate the task state to the target node,
(3) redirect the input streams to the target node,
(4) unquiesce all input streams.

4.3 Elasticity Handling

The same algorithm can be used to replace query tasks with their rewritten versions that compute the same logical transformation but use different operator implementations and/or partitioning schemes of the query graph into tasks for implementing the elasticity patterns 4 and 5.

Rewriting Cost Model: Usually, there are several possibilities for rewriting dataflow graphs. In order to make right decisions which tasks shall be replaced and how many nodes should be allocated, a cost model is required taking possible rewriting benefits into account as well as costs for the restructuring, e.g., for transferring states or costs for additional resources from the cluster infrastructure. Discussing elaborate decision models is not in the focus of this paper and is left for future work. We outline a rate-based model that is suitable to find hot spots in dataflows and is used in related literature [13].

Rewritings should be done when a query task is detected that cannot process its incoming tuples with a rate higher than the arrival rate, e.g., when the computational complexity or the memory consumption is too high and therefore the task accumulates an increasing backlog. Such a task represents the critical path in the dataflow, limiting the overall throughput. For finding these paths, a rate-based optimization approach is suitable that scans the dataflows starting at source nodes and detects such bottlenecks based on monitoring information gathered during the execution [13]. After hot spots have been identified, one or multiple of the following methods can be applied for alleviating these bottlenecks.

[2]Usually, the separation of λ and τ is only conceptual and both functions are combined.

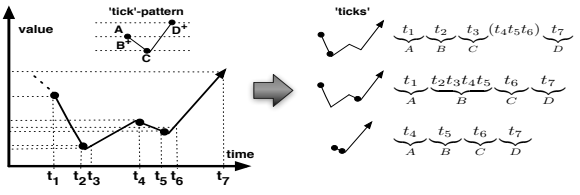


Figure 7: 'tick'-shaped pattern

Task sharing (Pattern 5): When multiple queries share the same (sub)graph to increase data locality [3] and the shared graph is on the critical path, this path can be replicated, sharing groups can be repartitioned, and tuples distributed to all replicas. This is the easiest way to remove burden from the critical path since inputs of sharing groups are independent from each other and no special dataflow transformations need to be performed.

Inter-operator parallelism (Pattern 4): Usually, it is better to keep dataflow operations on few nodes in order to avoid costly transfer operations. Hence, initially compiled queries will comprise few tasks consisting of large flow (sub)graphs. However, when the computational complexity exceeds a certain threshold or memory limits for keeping state information are exceeded, a distributed processing pipeline will yield better performance. Large graphs are partitioned, recompiled, and distributed on additional nodes in the cluster. Moreover, splitting large costly tasks into smaller distributed ones increases fault-tolerance since it will become cheaper to recover from node failures [11].

Intra-operator parallelism (Pattern 4+5): When the previous patterns not applicable, e.g., when operators are not shared or the graph has already been split into base operators, the last possibility to increase parallelism is partitioning input streams and processing each partition independent from each other with multiple operator instances. Unfortunately, this pattern is the most difficult to implement since its applicability depends on the actual operator type. Partitions in input streams need to be found, distributed to the operator instances, and their (partial) results have to be merged afterwards. Further, this pattern is prone to data skews and, therefore, some sort of load balancing has to be implemented, e.g., by monitoring the load of each partition and dynamically re-schedule partitions as proposed in [10]. This concept seamlessly integrates with the graph rewriting patterns described in this paper but again involves additional costs for transferring partitions among cluster nodes.

The most challenging problem in this context is finding suitable partitioning schemes for the operators that shall be deployed in the framework, especially when they are not stateless. In the following, we will present a parallelization scheme in the complex event processing (CEP) context which is prominent in stream processing.

The task of CEP is finding complex patterns in a stream of base patterns. These patterns are defined through certain properties of incoming tuples, usually described through predicates and additional correlations of their arrival time. Mostly, sequences and repetitions are used which can be expressed with regular expressions [15]. We demonstrate the parallelization on the 'tick-shaped' pattern example from [5] which is illustrated in Fig. 7. In the original publication, the task for detecting such patterns is originated in stock exchange trading, but it could also be applied for burst detection. A 'tick-shape' can be expressed by AB^+CD^+ where:

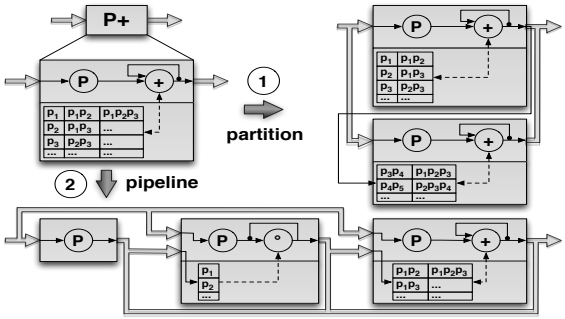


Figure 8: rewriting repetition operator

- A matches any incoming tuple
- B^+ matches all following tuples with decreasing value
- C matches the first tuple with increasing value after B but with a value less than the previous one of A
- D^+ matches following tuples with increasing values $> A$

For parallelization, we focus on the +-operator since it is challenging for three reasons: First, like a join, it can produce multiple output tuples per input tuple. Second, it needs to store state information for extending existing patterns to longer ones. Since each tuple is possibly multiplying the number of results it is likely that – due to memory constraints – such operators will become critical in the dataflow graph. Third, in most cases the behavior of the stream is not predictable, rendering static allocations infeasible.

Fig. 8 illustrates how the operator can be distributed dynamically to multiple nodes with simply applying graph patterns 4 and 5. The P+-operator comprises two parts: a pattern matcher P, and a +-operator which maintains all previously matched patterns as state and concatenates them with subsequent matches. The output of + is the output for the entire operator and serves as input for + again to construct longer matches. On memory overflow, the operator state can be **partition**ed and distributed to multiple instances where all instances receive the original input stream. Two different behaviors of the operator are required to avoid duplicates. The first instance processes the input directly, i.e., all matching tuples serve as new patterns of length 1. Those matches must not be reproduced by other instances that simply serve as targets for overflowing patterns that do not fit into the local state but are independent from each other and hence can be processed on separate nodes. When the complexity of the matching algorithm P is critical, elasticity can also be achieved with implementing a **pipeline**. It exploits the fact that P^+ can be expressed through $P \vee (PP^+)$, i.e., a pipeline of arbitrary length is constructed for matching incoming tuples in parallel, emitting them as output, and forwarding them to the next stage for extension.

Since such parallelization schemes depend on operator semantics, the framework provides them to automatically scale up and down required resources for built-in operators. For all user defined functions which are treated as black boxes by AnduINv2, the parallelization needs to be implemented by the user as in [11] or are provided through libraries that are linked as plugin to the execution environment.

4.4 Mesos Integration

Mesos [7] is a cluster management software for resource isolation and sharing. In Mesos, a master daemon (possibly supported by additional standby masters) manages a set of slaves nodes. An application (called framework) runs

tasks on these slaves which is initiated by so-called executors. Scheduling and resource assignments are managed by an application-specific scheduler. In order to support stream queries we implemented our own framework (cf. Fig. 6), providing an executor for running query executables (query tasks) on slave nodes and a *query scheduler* which gets resource offers from the Mesos master (available cores, memory, and network ports) and requests for executing AnduIN queries. Each query deployment request is described by a unique ID, the executable, and a specification of resource requirements, i.e., CPU cores, memory, and a list of query channels which have to be mapped to network ports. This specification is used by the scheduler to choose a slave node providing the requested resources for execution. Currently, only a simple strategy is implemented selecting the first offer providing the requested resources – more advanced strategies are subject of future work. If the scheduler has chosen an appropriate node, the request is forwarded to the corresponding executor. The scheduler assigns physical network ports to query channels and tracks these assignments to be able to connect subsequent queries referring to the same logical channel. In this way, a query implemented by one or more tasks can be deployed to one or more cluster nodes.

4.5 Meta Queries

Though, Mesos provides mechanisms to deploy processes, it does not support elastic operation for stream queries. In Hadoop, it is the task of the job tracker to partition the work across a set of map and reduce tasks. In case of data streams the situation is a bit different, because we cannot simply stop and continue/restart queries without loosing data. The only way to achieve elasticity is to change query behavior at runtime. Therefore, we introduce the idea of meta queries: in each (adjustable) query task an additional query is running on a control stream consisting of tuples of the form:

$$\langle \mathsf{query\_id}, \mathsf{operator\_id}, \mathsf{parameter}, \mathsf{value} \rangle$$

The control stream is produced by the query scheduler which monitors resource utilization and implements strategies for dynamic reallocation. Meta queries are particular useful for implementing the patterns described in Sect. 3. For instance, for failover without publish-subscribe (pattern 1 and 2), the network writer has to be informed about the network address of the newly activated standby node S. For this purpose, the network writer provides a parameter target-addr for the target address. A control stream tuple like

$$\langle \mathsf{query\#42}, \mathsf{writer\#2}, \mathsf{target\text{-}addr}, \mathsf{"tcp://node2:6666"} \rangle$$

received by the query task triggers sending the tuple stream to the standby node node2. Similarly, for implementing pattern 3, the stream selector node can be informed about switching to the stream produced by query node S.

For partitioning patterns like pattern 4 it is either required to modify the tuple distribution strategy of multicast writers or to adjust partitioning predicates P_i in Fig. 3. Both can be easily implemented by sending appropriate control tuples.

5. CONCLUSION AND FUTURE WORK

We presented basic concepts how fault-tolerance and elasticity can be achieved in the context of continuous query processing by combining techniques that have proven applicability in other scenarios. These approaches are currently being integrated into AnduINv2, but can be applied in other platforms, too. Our main questions we would like to answer with future experiments are:

1) Which cost models are valid for online graph rewriting?
2) How can resource requirements for a query be estimated before actually executing it?
3) How can certain QoS guarantees be given to applications?
4) Can elastic stream processing benefit from heterogeneous clusters nodes?

While the first questions intent to pave the way for a streaming-as-a-service infrastructure, answering the last one is needed to keep up with current hardware development trends. We believe that parallel and specialized processors as many-core CPUs, GPUs, or FPGAs will find their way into future computing centers to provide the most efficient computing platforms for dedicated tasks – an important aspect to tackle the big data challenge.

6. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, et al. The Design of the Borealis Stream Processing Engine. In *CIDR '05*, 2005.

[2] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*. Springer, 2004.

[3] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. *SIGMOD Rec.*, 29:379–390, 2000.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, pages 21–21, 2010.

[5] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *DEBS '11*. ACM, 2011.

[6] M. Eckert, F. Bry, S. Brodt, O. Poppe, and S. Hausmann. A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed. In *Reasoning in Event-Based Distributed Systems*. Springer, 2011.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, pages 22–22, 2011.

[8] S. Hirte, E. Schubert, A. Seifert, S. Baumann, D. Klan, and K. Sattler. Data3 - A Kinect Interface for OLAP using Complex Event Processing. In *ICDE*, 2012.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 41:59–72, 2007.

[10] M. S. Joseph, J. M. Hellerstein, S. Ch, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2002.

[11] Q. Ke, M. Isard, and Y. Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, pages 15–28, 2013.

[12] D. Klan, M. Karnstedt, K. Hose, L. Ribe-Baumann, and K. Sattler. Stream engines meet wireless sensor networks: cost-based planning and processing of complex queries in AnduIN. *Distrib. and Parallel Databases*, 29:151–183, 2011.

[13] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD '02*. ACM, 2002.

[14] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud '12*. USENIX Association, 2012.

[15] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern matching in sequences of rows. Technical report, ANSI Standard Proposal, 2007.

Task Graphs of Stream Mining Algorithms

Sayaka Akioka
Meiji University
4-21-1 Nakano, Nakano-ku
Tokyo, 164-8525, Japan
+81-3-5343-8305
akioka@meiji.ac.jp

ABSTRACT

Acceleration of huge data analysis, especially an analysis of huge, and fast streaming data is one of the major issues in recent computer science. Proper modeling, and understanding of streaming data analysis are indispensable for speed-up, scale out, and faster response time of streaming data analysis. Especially for the research on scheduling, or load balancing algorithms, a model of the target application truly impacts on the performance of the scheduling, or load balancing algorithms, however, there is no study on the realistic models, or the actual behaviors of streaming data analysis yet. This paper proposes a task graph for stream mining algorithms with some examples of actual applications. A task graph represents a workload of the target application with data dependencies, and control flows. This is the first proposal of task graphs for stream mining algorithms, and the task graphs play an important role as a benchmarking tool for the development of scheduling, or load balancing algorithms targeting on stream mining algorithms.

1. INTRODUCTION

Applications to process a massive amount of data, so-called "big data", is one of the recent hot research topics. Big data applications are sometimes considered to be quite similar with data intensive applications in high performance computing (HPC), however, the behaviors of applications in these two domains are quite different [9].

Big data applications utilize often stream mining algorithms, while data intensive applications process huge data in a batch. That is, big data application often tries to analyze data stream, which is a sequence of data arriving in chronological order, on the fly. As the data stream flows very fast, stream mining algorithms are developed with the purpose of the perfect analysis over such fast data flows. Once the delay of the analysis arises, and the analysis fails to keep up with the data arrival, the whole process will be forced to drop some of the arriving data. As many of the streaming analysis processes place emphasis on the real-time analysis in chronological order, a drop of the arrival data is highly critical.

As big data applications scale up with such a severe requirement for extremely low latency, big data applications become to run on the parallel and distributed computing environment such as the computing cloud. In order to exploit parallelism, and speed up the applications, scheduling is indispensable. Scheduling algorithms in parallel and distributed computing environment have been studied intensively for a long time especially in HPC, and these researches often validate, and compare the scheduling algorithms with task graphs. A task graph represents a workload of a target application, which is often synthetic workload generated randomly. As the quality of task graphs heavily impacts on validation of scheduling algorithms, the methodology to generate task graphs have been studied as well with a strong focus on data intensive applications in HPC.

This paper proposes task graphs generated from the actual implementations of stream mining algorithms in order to contribute to a development of effective, and practical scheduling algorithms for stream mining algorithms. The contributions of this paper are 1) the first proposal of task graphs for stream mining algorithms, 2) the practical and realistic workloads extracted from the existing implementations, 3) task graphs as representations of the behaviors of stream mining algorithms to open up unexplored problems for conventional scheduling algorithms, and 4) task graphs as a benchmarking tool to accelerate the development of scheduling algorithms for stream mining algorithms.

The rest of this paper is organized as follows. Section 2 gives a generic model of stream mining algorithms in order to clarify data dependencies of the process. Section 3 describes the procedure of task graph generation, and proposes a format of task graphs for stream mining algorithms. Section 4 overviews actual stream mining algorithms analyzed in this paper, and represents corresponding task graphs. Section 5 briefly introduces the related work, and Section 6 concludes this paper.

2. STREAM MINING ALGORITHMS

A stream mining algorithm is an algorithm specialized for a data analysis over data streams on the fly. There are many variations of stream mining algorithms, however, general stream mining algorithms share a fundamental structure, and a data access pattern as shown in Figure 1 [1].

A stream mining algorithm consists of two parts; a stream processing part, and a query processing part. First, the stream processing module picks the target data unit, which is a chunk of data arrived in a limited time frame, and executes a quick analysis over the data unit. The quick analysis here can be a preconditioning process such as a morphological analysis, or a word counting. Second, the stream processing module updates the data cached in one or more sketches with the latest results through the quick analysis. That is, the sketches keep the intermediate analysis, and the stream processing module updates the analysis incrementally as more data units are processed. Third, the analysis module reads the intermediate analysis from the sketches, and extracts the essence of the data in order to complete the quick analysis in the stream processing part. Finally, the query

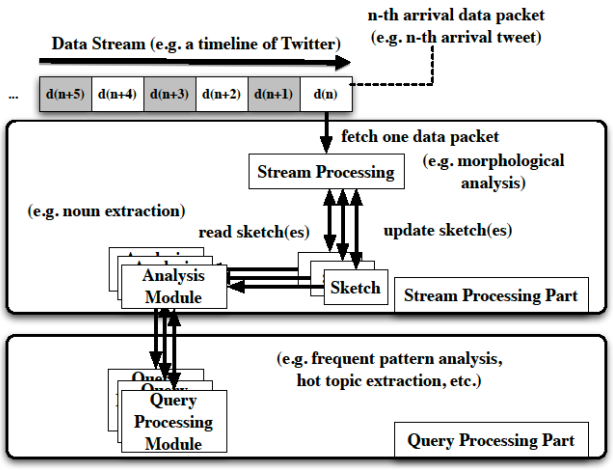


Figure 1. A model of stream mining algorithms.

processing part receives this essence for the further analysis, and the whole process for the target data unit is closed.

Based on the modeling above, we can conclude that the major responsibility of the stream processing part is to process each data unit for the further analysis, and that the stream processing part has the huge impact over the latency of the whole process. The stream processing part needs to finish the preconditioning of the current data unit before the next data unit arrives, otherwise, the next data unit will be lost as there is no storage for buffering the incoming data in a stream mining algorithm. On the other hand, the query processing part takes care of the detailed analysis such as a frequent pattern analysis, or a hot topic extraction based on the intermediate data passed by the stream processing part. The output by the query processing part is usually pushed into a database system, and there is no such an urgent demand for an instantaneous response. Therefore, only the stream processing part needs to run on a real-time basis, and the successful analysis over all the incoming data simply relies on the speed of the stream processing part.

The model of a stream mining algorithm shown in Figure 1 also indicates that the data access pattern of the stream mining algorithms is totally different from the data access pattern of so-called data intensive applications, which is intensively investigated in HPC. The data access pattern in the data intensive applications is a write-once-read-many [9]. That is, the application refers to the necessary data many times during the computation; therefore, the key for the speedup of the application is to place he necessary data close to the computational nodes for the faster data accesses. On the other hand, in a stream mining algorithm, a process refers to its data unit only once, which is a read-once-write-once style. Therefore, a scheduling algorithm for the data intensive applications is not simply applicable or the purpose of the speedup of a stream mining algorithm.

Figure 2 illustrates data dependencies between two processes analyzing data units in line, and data dependencies inside ne process. The left top flow represents the stream processing part of the preceding process, and the right bottom flow represents the stream processing part of the successive process. Each flow consists of the six stages; read from sketches, read from input, stream processing, update sketches, read from sketches, and analysis. An arrow represents a control flow, and a dashed arrow represents a data dependency.

In Figure 2, there are three data dependencies in total as follows, and all of these three dependencies are essential to keep the analysis results consistent, and correct.

1. The processing module in the preceding process should finish updating the sketches before the processing module in the successive process starts reading the sketches (Dep.1 in Figure 2).
2. The processing module should finish updating the sketches before the analysis module in the same process starts reading the sketches (Dep.2 in Figure 2).
3. The analysis module should finish reading the sketches before the processing module in the successive process starts updating the sketches (Dep.3 in Figure 2).

3. TASK GRAPH DEFINITIONS

As discussed in Section 2, a model of a stream mining algorithm has data dependencies both across the processes, and inside one process. Therefore, a task graph or a stream mining algorithm should consist of a data dependency graph, and a control flow graph. We already modeled both the data dependencies, and the control flows for stream mining algorithms in Section 2, however, a task graph is a finer grained model for a specific algorithm and implementation.

A data dependency graph is drawn via an analysis of the actual implementation of the target algorithm. Figure 3(a) is an example of a data dependency graph of the training stage of Naïve Bayes classifier[2] implemented by MOA project [8]. Figure 4 represents a pseudo code for the data dependency graph in Figure 3(a). A data dependency graph is a directed acyclic graph (DAG). In a data dependency graph, each node represents a basic block, or

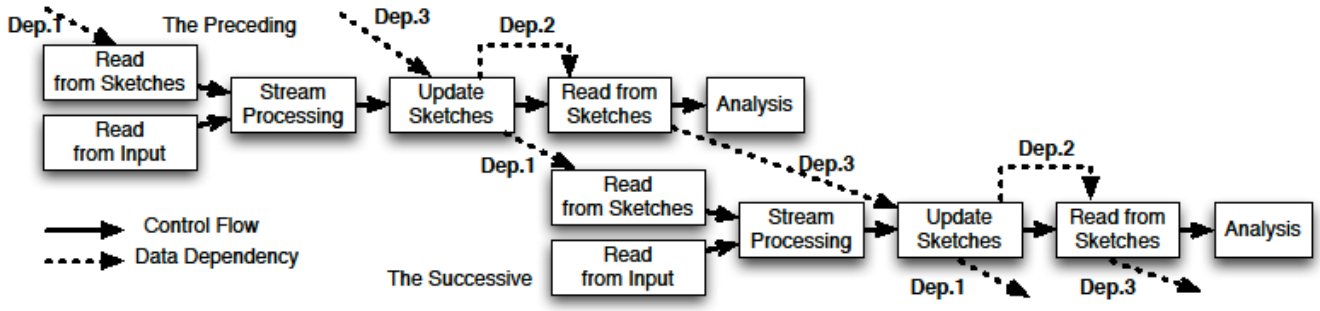


Figure 2. Data dependencies of the stream processing parts in two processes in line.

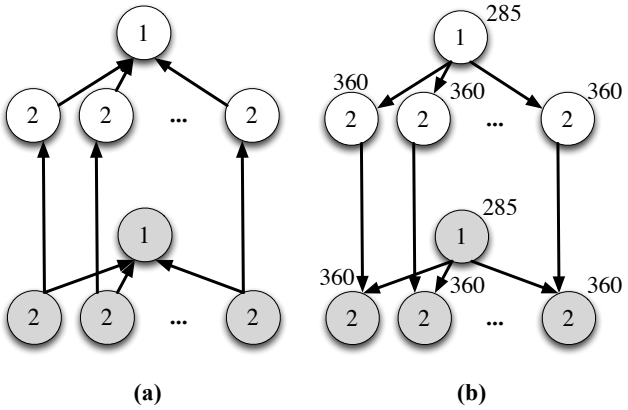


Figure 3. The data dependency graph (a), and the control flow graph (b) for Naïve Bayes implementation of MoA.

for all training data **do**
  (1) Fetch one training data *v*
  for all attributes for *v* **do**
    (2-1) Update the weight sum of this attribute.
    (2-2) Update the mean value of this attribute.
  end for
end for

Figure 4. The training stage of Naïve Bayes algorithm.

an equivalent chunk of codes in the actual implementation, and each array indicates a data dependency. If an arrow comes up from node A to node B, the arrow indicates that there is a data dependency between node A, and node B, and that the process represented by node B relies on the data generated by the process represented by node A for consistency of the analysis.

A data dependency graph in Figure 3(a) actually consists of two DAGs; a DAG with nodes in white, and a DAG with nodes in gray. Each DAG represents each process in Figure 2. That is, the DAG with white nodes in Figure 3(a) indicates the preceding process in Figure 2, and the DAG with gray nodes in Figure 3(b) indicates the successive process in Figure 2. The arrows between the two DAGs represent data dependencies between the two processes. In the case of stream mining applications, which is the most different point from conventional applications, the application continues running as long as a new data unit arrives. A DAG with nodes in a same color represents one process for one data unit, therefore, DAGs should lie in a line as many as the number of data the corresponding application processes. In this case, two DAGs are sufficient for the representation of the minimum unit of the repeated pattern in the application, and the data dependency graph does not contain any more redundant DAGs for simple but sufficient representation.

In a data dependency graph, each node has a number, and the number indicates that the particular node represents which basic block in the pseudo code, such as shown in Figure 4. In this example, node 1 represents the line starting with "(1)" in the pseudo code in Figure 4, and node 2 represents the lines starting

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="NodeType">
  <xs:attribute name="id" type="xs:string" use="required" />
  <xs:attribute name="cost" type="xs:int" use="required" />
  <xs:attribute name="parallelism" type="xs:int" />
 </xs:complexType>

 <xs:complexType name="ArrowType">
  <xs:attribute name="id" type="xs:string" use="required" />
  <xs:attribute name="src" type="xs:string" use="required" />
  <xs:attribute name="dest" type="xs:string" use="required" />
 </xs:complexType>

 <xs:complexType name="DummyNodeType">
  <xs:attribute name="id" type="xs:string" use="required" />
  <xs:attribute name="cost" type="xs:int" fixed="-1" />
 </xs:complexType>

 <xs:complexType name="DDType">
  <xs:sequence maxOccurs="unbounded" minOccurs="2">
   <xs:element ref="arrow" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="CFType">
  <xs:sequence maxOccurs="unbounded" minOccurs="2">
   <xs:element ref="arrow" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="NodeListType">
  <xs:sequence>
   <xs:element ref="startNode" />
   <xs:sequence maxOccurs="unbounded" minOccurs="1">
        <xs:element ref="node" />
   </xs:sequence>
   <xs:element ref="endNode" />
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="TaskGraph">
  <xs:sequence>
   <xs:element ref="nodeList" />
   <xs:element ref="cf" />
   <xs:element ref="dd" />
  </xs:sequence>
 </xs:complexType>

 <xs:element name="node" type="NodeType" />
 <xs:element name="startNode" type="DummyNodeType" />
 <xs:element name="endNode" type="DummyNodeType" />
 <xs:element name="arrow" type="ArrowType" />
 <xs:element name="dd" type="DDType" />
 <xs:element name="cf" type="CFType" />
 <xs:element name="nodeList" type="NodeListType" />
 <xs:element name="taskgraph" type="TaskGraph" />
</xs:schema>
```

Figure 5. XML scheme for a task graph.

```
<?xml version="1.0" encoding="utf-8" ?>
<taskgraph>
  <nodeList>
    <startNode id="0" />
    <node id="1" cost="285" parallelism="1" />
    <node id="2" cost="360" parallelism="-1" />
    <endNode id="3" />
  </nodeList>

  <cf>
    <arrow id="cfa01" src="0" dest="1" />
    <arrow id="cfa12" src="1" dest="2" />
    <arrow id="cfa22" src="2" dest="2" />
    <arrow id="cfa23" src="2" dest="3" />
  </cf>

  <dd>
    <arrow id="dda10" src="1" dest="0" />
    <arrow id="dda21" src="2" dest="1" />
    <arrow id="dda22" src="2" dest="2" />
    <arrow id="dda32" src="3" dest="2" />
  </dd>
</taskgraph>
```

Figure 6. XML representation for the task graph in Figure 3.

with "(2-1)", and "(2-2)" in the pseudo code in Figure 4. Here, as determined from the pseudo code in Figure 4, the basic block indicated by node 2 is data parallel. Therefore, in the data dependency graph, several node 2s are located in the same level of the DAG. Logically, there is no limit of the number of node 2s in this case, therefore, an user can put node 2s as many as desired.

A control flow graph is also drawn via an analysis of the actual implementation of the target algorithm again, and the basic definitions are almost the same to the case of a data dependency graph. Figure 3(b) is a control flow graph for Naïve Bayes classifier, and the corresponding pseudo code is shown in Figure 4. Each node represents basic block again, however, each arrow in a control flow graph represents the order of the process of basic blocks. That is, in Figure 3(b), node 2 always has to be processed just after node 1 is completed. On the other hand, nodes without arrays in between do not have any ordering restriction. Therefore, these nodes can be executed in a shuffled order, or on the same stage. As the same to the data dependency graph, a control flow graph consists of the minimum but sufficient DAGs for the simplicity.

A control flow graph has a computational cost for each node. A computational cost shown in a control flow graph is the average of the actual computational costs measured in the actual computations, however, this version of the control flow graphs do not contain communication costs. As the control flow graphs here are fine-grained, it is not beneficial to scatter one control flow graph over the distributed computing environment. That is, pipelining control flow graphs according to the speed of the input data is a more realistic, and practical solution. Communication costs for pipelining in the distributed computing environment contains further discussions, and we reserve this topic for the future work.

Figure 5 is the XML schema for a task graph, and Figure 6 is an example representation of XML for Figure 3. Task graphs should be represented also in XML according to this schema, and a designer of scheduling simulators can easily employ the task graph as a benchmark by reading this XML.

for all input data items **do**
  (1) fetch one input data v
  for all distinct items appeared **do**
    (2) create or update a border point for v
    (3) update summary
    (4) update frequency
    (5) delete obsolete border points
  end for
  (6) update pruning threshold
end for

Figure 7. A pseudo-code of top-k (min summary).

4. ACTUAL TASK GRAPHS

This section introduces task graphs extracted from the actual popular methodologies. One is top-k implemented as a Java 1.7 application. The other is Hoeffding tree algorithm[6], which is one of decision tree algorithms, and implemented as MOA module[8].

We implemented top-k based on min summary algorithm proposed by Lam et al.[7], and the base proposal by Calders et al. [3]. Figure 7 is the pseudo-code of the corresponding algorithm. Figure 8 (a) represents the extracted data dependency graph, and Figure 8(b) represents the extracted control flow graph.

As we already saw through the generic model of the stream mining algorithms in Section 2, each node processing one data unit basically depends on the results of the previous node. That is, each node is a consumer of the previous node. The exception is node 1 (data fetching), and node 6 (update of the pruning threshold). Especially, node 6 updates the pruning threshold based on the length of the summary, and node 6 has to wait for the elimination of the obsolete border points, which is node 5. On the other hand, the process represented from node 2 to node 5 is independent across the distinct items appeared during the observation, and this part is capable of parallel execution.

When we focus on the dependency between the preceding process, and the successive process, node 2 in the successive process depends on node 5 in the preceding process. Node 5 in the preceding process deletes the obsolete border points, while node 2 adds a new border point, or increment the counter of the existing border point according to the input. There is no dependency when node 2 adds a new border point, however, node 2 needs to decide which border point should be updated when node 2 increments the count of the existing border point. This is the reason why node 2 in the successive process behaves as a consumer of node 5 in the preceding process.

One more thing we would note here is that the computational cost of node 6 is relatively huge compared to the computational costs of the other nodes. The major reason of the heavy load of node 6 is that node 6 needs to calculate the maximum relative frequency of the least appeared item during the observation. Because of this process, node 6 is a consumer of node 5, needs to sweep all the data in the summary, and consumes more time for completion.

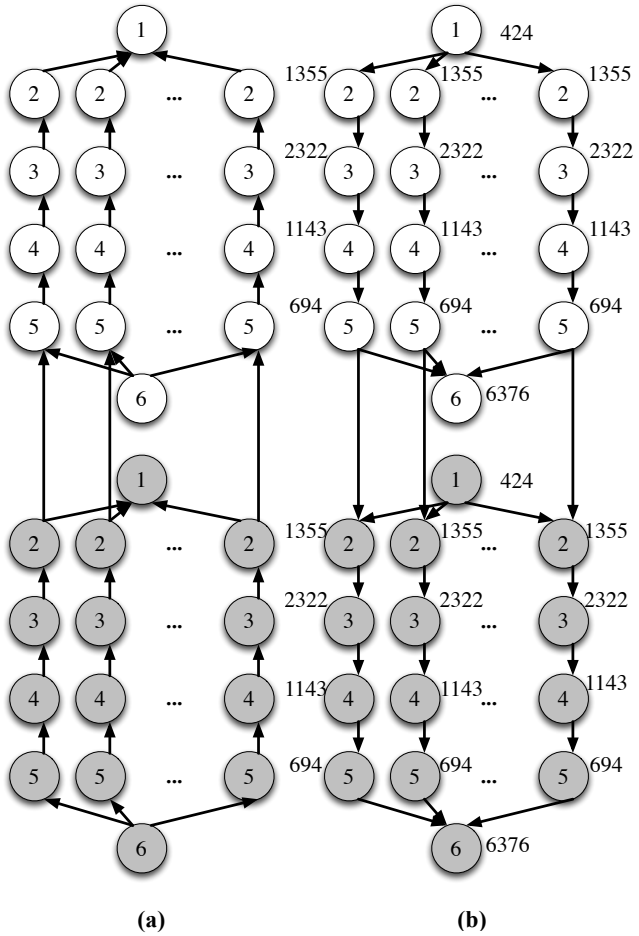


Figure 8. The data dependency graph (a), and the control flow graph for top-k (min summary).

This tendency of the computational cost implies that the execution in a pipeline is really effective for min summary algorithm. In fact, the computational cost of node 6 is almost equivarent to the total cost of the single path of nodes 1-5, and node 6 is independent from these nodes. Therefore, node 1-5, and node 6 are capable of running in a pipeline, and consumes almost the same computational time. That is, there is a chance to hide almost half of the execution time of the process time of one data unit, and improve the throughput by pipelining.

We also extracted a task graph of Hoeffding tree algorithm from MOA implementation. Figure 9 is the pseudo-code of the algorithm, and Figure 10 represents the extracted data dependency graph. The control graph is omitted for the page limitation. We skip the detailed discussion for the page limitation again, however, we can observe similar tendency of the application as we saw in the generic model in Section 2, Naïve Bayes in Section 3, and min summary algorithm in this section. One major difference from the previous cases is that node 1 depends on node 9, therefore, the effect of the pipelining is not huge compared to the other cases.

Here, we need to discuss computational costs in the control flow graph. This version of the task graph represents a computational cost as the average of actual executions. This is in a sort of the simplified model as the computational cost of stream mining algorithms easily varies depending on the input data. We need to develop the better methodology for the computational model, however, we reserve this issue as a future work.

```
Let HT be a tree with a single leaf (the root)
for all training data do
   (1)  Fetch one training data v, and sort v into leaf l
        using HT
   for all attributes for v do
      (2) Update sufficient statistics in l
   end for
   (3) Increment nl, the number of examples seen at l
   if n1 mod nmin = 0 and data seen at l not all of same
   class then
      (4) Compute Gl(Xi) for each attribute
      (5) Let Xa be attribute with highest Gl
      (6) Let Xb be attribute with second-highest Gl
      (7) Compute Hoeffding bound
       if Xa != Xb and (Gl(Xa) − Gl(Xb) > ε  or  ε  < τ)
       then
         (8) Replace l with an internal node that splits on Xa
         for all branches of the split do
            (9) Add a new leaf with initialized sufficient
                statistics
         end for
      end if
   end if
end for
```

Figure 9. A pseudo-code of a training tree of Hoeffding tree algorithm.

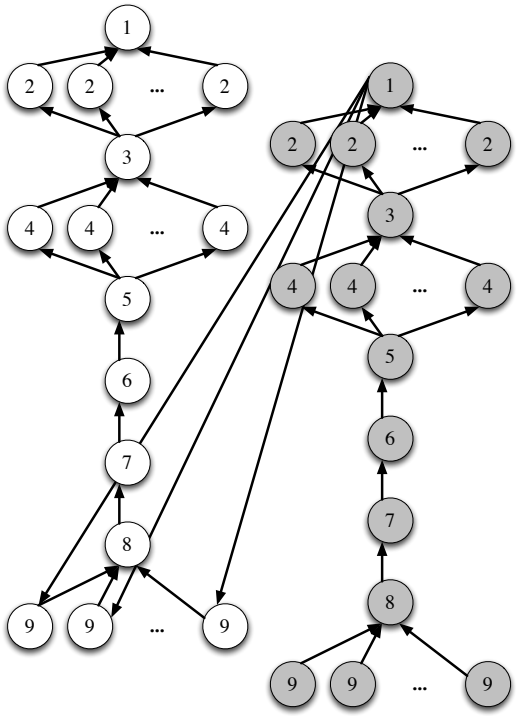


Figure 10. A data dependency graph for Hoeffding tree algorithm.

5. RELATED WORK

There are several studies on task graph generation, mainly focusing on random task generation. A few projects reported task graphs generated based on the actual well-known applications, however, those applications are from numerical applications such as Fast Fourier Transformation, or applications familiar to HPC community for a long time.

Task Graphs for Free (TGFF) provides pseudo-random task graphs [5,11]. TGFF allows users to control several parameters, however, generates only directed acyclic graphs (DAGs) with one or multiple start nodes, and one or multiple sink nodes. Each task graph is assigned a period, and a deadline based on the length of the maximum path in the graph, and the user specified parameter.

GGen is another random task graph generator proposed by Cordeiro et al [4]. GGen generates random task graphs according to the well-known random task generation algorithms. In addition to the graph generator, GGen provides a graph analyzer, which characterizes randomly generated task graphs based on the longest path, the distribution of the out-degree, and the number of edges.

Task graph generator provides both random task graphs, such as Fast Fourier Transformation, Gaussian Elimination, and LU Decomposition [12]. The random task graph generator supports variety of network topologies, including star, and ring. Task graph generator also provides scheduling algorithms as well.

Tobita et al. proposed Standard Task Graph Set (STG), evaluated several scheduling algorithms, and published the optimal schedules for STG [10,13]. STG is a set of random task graphs, which are ready to download. Tobita et al. also provides task graphs from numerical applications such as a robot control programs, a sparse matrix solver, and SPEC fpppp.

Besides the studies on task graph generation, Cordeiro et al. pointed out that randomly generated task graphs can create biased results, and that the biased results can mislead the analysis of scheduling algorithms[4]. According to the experiments by Cordeiro et al., a same scheduling algorithm man obtain a speedup of 3.5 times only by changing the graph generation algorithm for the performance evaluation.

Random task graphs contributes positively for evaluation of scheduling algorithms, however, do not perfectly cover all the domains of parallel and distributed applications as Cordeiro et al. figured out in their work. Especially for stream mining applications, which focus on in this paper, the characteristic of the application behaviors are quite different from the characteristic of the applications familiar to the conventional HPC community as we discussed in Section 2. Task graphs generated from the actual stream mining applications have profound significance in the better optimization of the applications in parallel computing environment for wider area of applications.

6. CONCLUSION

This paper proposed task graphs for stream mining algorithms. This is the first clear proposal of task graphs modeling stream mining algorithms, and the task graphs are extracted from the actual implementations of the popular existing methodologies. Task graphs proposed in this paper play an important role as the benchmarking tool to evaluate scheduling algorithms, or load balancing algorithm, which is indispensable for the research of scheduling, or load balancing algorithms truly effective for stream mining algorithms. In fact, in this paper, the proposed task graphs represent apparently different characteristics, and dependencies compared to the data intensive applications in HPC, and this fact points out we need to consider scheduling methodologies focusing on stream mining algorithms. For the better set of task graphs, we are working on more stream mining algorithms.

7. REFERENCES

[1] Akioka, S., Muraoka, Y., Yamana, H., Data access pattern analysis on stream mining algorithms for cloud computation. In *Proceedings of the 2011 International Conference on Parallel and Distributed Processing (PDPTA2011)* (Las Vegas, USA, July 18-21, 2011), 2011, 36-42.

[2] Bifet, A., Holmes, G., Pfahringer, B., Karen, P., Kremer, H., Jansen, T., Seidl, T., MOA: Massive online analysis, a framework for stream classification and clustering. *Journal of Machine Learning Research (JMLR)*, Workshop and Conference Proceedings Vol. 11: Workshop on Application of Pattern Analysis, 2010.

[3] Calders, T., Dexters, N., Goethals, B., Mining Frequent Itemsets in a Stream. In *Proceedings of 2007 IEEE International Conference on Data Mining* (ICDM2007) (Omaha, USA, October 28-31, 2007), 2007.

[4] Corderiro, D., Mounie, G., Perarnau S., Trystram, D., Vincent, J. M., Wagner, F., Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools'10)* (Torremolinos, Spain, March 15-19, 2010), 2010.

[5] Dick, R. P., Rhodes D. L., Wolf, W., TGFF: Task graphs for free. In *Proceedings of International Workshop on Hardware/Software Codesign* (Seattle, USA, March 15-18, 1998), 1998, 97-101.

[6] Domingos, P., Hulten, G., Mining High-Speed Data Streams. In Proceedings of *The 6th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD'00) (Boston, USA, August 20-23, 2000), 2000, 71-80.

[7] Lam, H. G., Calders, T., Mining top-k frequent items in a data stream with flexible sliding window. In *Proceedings of The 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (KDD'10) (Washington DC, USA, July 25-28, 2010), 2010.

[8] McCallum A., Nigram, K., A comparison of event models for Naïve Bayes text classification. In *Proceedings of AAAI-98 Workshop on 'Learning for Text Categorization'* (Madison, USA, July 26-27, 1998), 1998.

[9] Raicu, I., Foster, I. T., Zhao, Y., Little, P., Moretti, C. M., Chaudhary, A., Thain, D. The quest for scalable support of data intensive workloads in distributed systems. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC2009)* (Munich, Germany, June 11-13, 2009), 2009, 207-216.

[10] STG, Standard task graph set. http://www.kasahara.elec.waseda.ac.jp/schedule/index.html.

[11] TGFF. http://ziyang.eecs.umich.edu/~dickrp/tgff/.

[12] TGG, Task graph generator. http://taskgraphgen.sourceforge.net/.

[13] Tobita T., Kasahara, H., A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, Volume 5, Issue 5, 2002, 379-394.

Large-scale Online Mobility Monitoring with Exponential Histograms

Christine Kopp
Fraunhofer IAIS
St. Augustin, Germany
christine.kopp@iais.fraunhofer.de

Michael Mock
Fraunhofer IAIS
St. Augustin, Germany
michael.mock@iais.fraunhofer.de

Odysseas Papapetrou
Technical University of Crete
Chania, Greece
papapetrou@softnet.tuc.gr

Michael May
Fraunhofer IAIS
St. Augustin, Germany
michael.may@iais.fraunhofer.de

ABSTRACT

The spread of digital signage and its instantaneous adaptability of content challenges out-of-home advertising to conduct performance evaluations in an online fashion. This implies a tremendous increase in the granularity of evaluations as well as a complete new way of data collection, storage and analysis. In this paper we propose a distributed system for the large-scale online monitoring of poster performance indicators based on the evaluation of mobility data collected by smartphones. In order to enable scalability in the order of millions of users and locations, we use a local data processing paradigm and apply exponential histograms for an efficient storage of visit statistics over sliding windows. In addition to an immediate event centralization we also explore a hierarchical architecture based on a merging technique for exponential histograms. We provide an evaluation on the basis of a real-world data set containing more than 300 million GPS points corresponding to the movement activity of nearly 3,000 persons. The experiments show the accuracy and efficiency of our system.

1. INTRODUCTION

Advertising media are under the obligation to provide reliable performance indicators for the pricing of advertising campaigns. For the German out-of-home (OOH) advertising industry, generating yearly net sales of about 760 million Euro [2], this has meant to establish a system of geographically differentiating performance indicators over the past years[1]. However, with the spread of digital signage also a fine-grained *temporal* differentiation will be required in future. While current performance indicators inform about the poster contacts of seven or ten average days (the two standard durations of poster campaigns in Germany), digital out-of-home (DOOH) advertising spots have a duration of only a few seconds. Assuming an evaluation period of 10 seconds, the granularity of the performance indicators (and consequently of the required input data) increases by four orders of magnitude. DOOH therefore has to face the challenge of collecting and analyzing *big data*. In addition, digital content has the advantage that it can be instantly adapted to a changing audience. This adaptation, however, requires *online* performance information, which forms the second challenge of DOOH performance evaluation.

In this paper we propose a distributed system for the large-scale online monitoring of poster performance indicators based on the evaluation of mobility data collected by smartphones. We hereby consider two use cases which the system shall cover. First, we want to be able to perform online queries which obtain performance measures for the recent past in a sliding window style. Second, we want to analyze historic data for various time intervals. The first type of query allows the online monitoring of poster performance and thus the targeted placement of advertisement spots. The second type of query can be used for billing purposes or to analyze previously collected data sets (e.g. to find interesting visit patterns that can then be monitored in the online system). Although our use cases differ with respect to their system requirements (distributed online processing vs. analysis of massive amounts of centralized data), we want to keep the maintenance effort of the system as low as possible. Our goal is therefore to set up a scalable system architecture that allows an efficient re-use of code from the online scenario for historic data analysis.

The key component of our approach to handle massive streams of data is to use exponential histograms for data compression. This data structure has the advantage that it offers sliding window query capabilities with a guaranteed maximum relative error. In addition, exponential histograms can be applied in a distributed setting [12] thus allowing for scalability when the number of users increases.

Our online system relies on an Android implementation

[1]http://www.agma-mmc.de/media-analyse/plakat.html

that we have used in previous work [3] to detect visit patterns on mobile phones. For the analysis of historic data we have set up a Storm environment. In combination with the Kafka messaging system we are able to perform historic data analysis in a distributed streaming fashion. In this way we can apply the same system architecture for online and historic data analysis. We use the Storm/Kafka environment to perform the experiments in this paper.

We analyze the performance of our system using a real-world GPS data set containing trajectories of 2,967 persons containing more than 300 million GPS points over a period of one week. We extract visit events from this data set using 400,988 points of interest (POI) in Germany from OpenStreetMap (OSM). Our experiments show that the usage of exponential histograms results in an average error of less than 1/10 of the maximum acceptable error while reducing the storage space to an amount as small as 9.7% of the baseline storage space.

The remainder of our paper is organized as follows. Section 2 discusses related work. Section 3 shows our system architecture and Section 4 provides the experiments. We conclude our paper in Section 5.

2. RELATED WORK

2.1 Exponential Histograms

Exponential histograms [1] are a deterministic structure, proposed to address the basic counting problem, i.e., for counting the number of true bits in the last N stream arrivals. They belong to a family of methods that break the sliding window range into smaller windows, called buckets or basic windows, to enable efficient maintenance of the statistics. Each bucket contains the aggregate statistics, i.e., the number of arrivals and bucket bounds, for the corresponding sub-range. Buckets that no longer overlap with the sliding window are expired and discarded from the structure. To compute an aggregate over the whole (or a part of the) sliding window, the statistics from all buckets overlapping with the query range are aggregated. For example, for basic counting, aggregation is a summation of the number of true bits in the buckets. A possible estimation error can be introduced due to the oldest bucket inside the query range, which usually has only a partial overlap with the query. Therefore, the maximum possible estimation error is bounded by the size of the last bucket.

To reduce the space requirements, exponential histograms maintain buckets of exponentially increasing sizes. Bucket boundaries are chosen such that the ratio of the size of each bucket b with the sum of the sizes of all buckets more recent than b is upper bounded. In particular, the following invariant is maintained for all buckets j: $C_j/(2(1+\sum_{i=1}^{j-1} C_i)) \leq \varepsilon$ where ε denotes the maximum acceptable relative error and C_j denotes the size of bucket j (number of true bits arrived in the bucket range), with bucket 1 being the most recent bucket. Queries are answered by summing the sizes of all buckets that fully overlap the query range, and half of the size of the oldest bucket, if it partially overlaps the query. The estimation error is solely contained in the oldest bucket, and is therefore bounded by this invariant, resulting in a maximum relative error of ε.

Recently, Papapetrou et al. [12] showed how an arbitrary number of exponential histograms $EH_1, EH_2, ..., EH_n$ (each one corresponding to an individual stream) can be aggregated/merged, in order to produce a single exponential histogram $EH_\oplus$ that corresponds to the order-preserving union of the streams. More precisely, let ε denote the maximum error parameter of the original exponential histograms, and ε' the parameter of the merging algorithm. The algorithm supports the creation of an aggregated exponential histogram with a maximum relative error of $(\varepsilon + \varepsilon' + \varepsilon \cdot \varepsilon')$. In this work we use this merging algorithm to reduce the memory required for storing the exponential histograms of the visit events coming from various input sources.

2.2 Distributed Evaluation of Visit Events

In previous work we have provided a set of visit quantities that can be used to define performance measures in OOH advertisement [8]. In this paper we concentrate on the evaluation of *gross visits* which state the number of total visits to a certain location and which can be used to estimate the total contacts to a poster site. In addition, we have provided a methodology for the privacy-preserving, distributed collection of visit quantities in previous work [7].

The basic idea of the approach is to decentralize the data collection and evaluation process of movement data. Instead of constantly submitting location information of a user to some central server, the evaluation of visits (or visit patterns) is performed *locally* on a mobile device (e.g. smartphone). The device submits only aggregated and anonymized statistics to a central coordinator. In addition, web anonymization techniques such as onion routing [4] can be used to prevent that the coordinator reconstructs visit histories from several messages of a person based on the communication protocol. A similar, however analytically less powerful framework has previously been proposed by Hoh et al. [6] for the distributed, privacy-preserving monitoring of traffic. However, both papers do not consider the practical aspect of scaling the proposed method to thousands and potentially millions of users. In fact, considering movement statistics from our GPS data set, every person traverses more than 200 street segments per day. If we assume further that each person visits 10 different locations (e.g. work location, shops, bus stops) per day and 20 million persons participate in data collection, about 4.2 billion events occur every day. In order to cope with this number of events, sophisticated analysis and storage algorithms as well as a sophisticated system architecture have to be devised. The design and performance analysis of such a system is the scope of our paper.

3. SYSTEM ARCHITECTURE

Our architecture consists of two or alternatively three layers (see Figure 1). The lowest layer holds the user nodes, which collect the users' GPS data and extract visit events. The visit events are forwarded either directly to the central coordinator (flat setting) or to a layer of intermediate nodes (hierarchical setting). In the flat setting, the coordinator aggregates the visit information of each POI in an exponential histogram. I.e., for each POI an exponential histogram is maintained that records the visit events for this POI. As the exponential histogram stores a time aggregate with the event, queries over time windows can be answered. In the hierarchical setting, the exponential histograms reside already at the intermediate nodes. In regular time intervals the intermediate nodes submit the exponential histograms to the coordinator, which merges them and answers user queries.

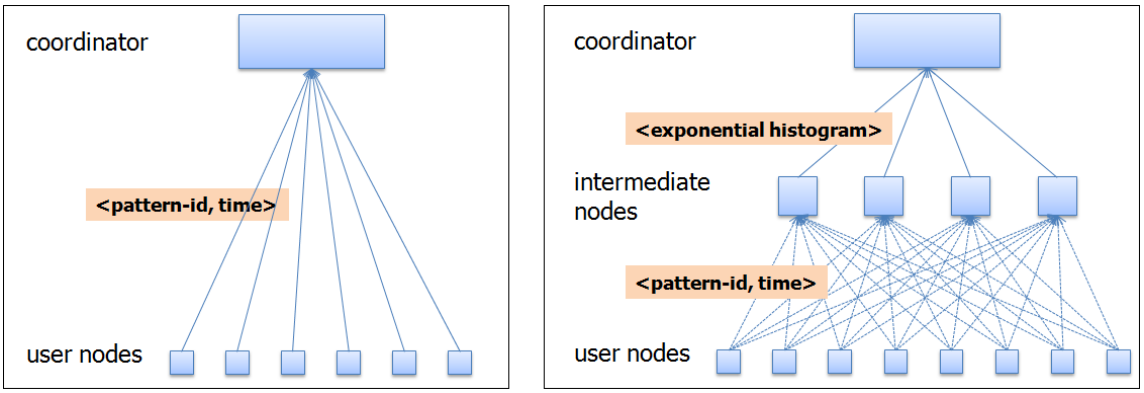


Figure 1: System architecture; left: flat setting; right: hierarchical setting

The exponential histograms cannot be applied at the user level because the number of visit events per user is too small to make the data structure efficient. The layer of intermediate nodes was introduced for horizontal scalability and to avoid an overload of the coordinator. However, it also serves a privacy purpose given that the intermediate nodes do not collude (see [7]). As a user can freely select an intermediate node when submitting a visit event, no intermediate node will obtain the whole event history of a single user. The intermediate nodes submit their data structure in regular time intervals to the coordinator, which finally merges the data structures and answers user queries.

As motivated by our use case, our system shall be able to perform analyses online as well as on historic data. The above architecture describes the online use case. For historic data analysis we have to substitute the layer of local nodes. This substitution should still allow to process data in parallel in order to scale to large amounts of data. In addition, a streaming environment would be preferable in order to re-use existing code. Both aspects can be met by using a distributed streaming processing system as, for example, Storm[2] or S4 [11]. We have ported the Android code of event detection to run as Storm bolts. The input is streamed into the system via the Kafka messaging system [9], which allows to handle each GPS point of the recorded trajectories as individual message. Thus, with this mechanisms we can scale the parallel simulation of event detection horizontally in the cluster. In our experiments described in the next section we used this technique to emulate the event detection on GPS traces of 2,967 test persons in an experimental cluster. Detected events are sent to the intermediate nodes similar to the online setting.

4. EXPERIMENTS

4.1 Data Set

For our experiments we use a subset of a large-scale GPS survey [10] commissioned by the Arbeitsgemeinschaft Media-Analyse e.V.[3], a joint industry committee of German advertising vendors and customers. The GPS data has been collected in the year 2011 and contains 2,967 persons with valid GPS data. The persons are recruited from 31 major cities in Germany and are asked to carry the GPS devices for one week.

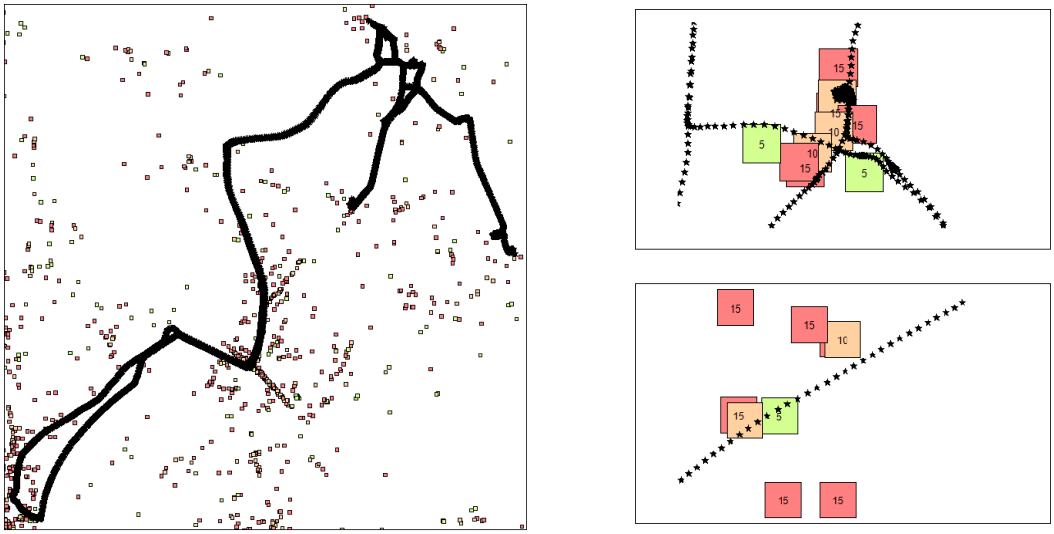
After clean-up the data set contains 304 million GPS points. In addition, we extracted 400,988 points of interest (POI) from OpenStreetMap[4] (OSM) [5] marked with the keys *shop, amenity, leisure, tourism, historic, sport, public transport, railway*. We grouped the POI into the following categories: shop, restaurant, leisure, education, parking and public transport stops. We limited our experiments to those POI because digital posters are still very expensive and therefore placed mostly at attractive places as train stations or shopping locations. For each POI category we defined a minimum stay time and a 50x50 meter spatial buffer in order to extract visit events. Table 1 shows the number of POI aggregated to the six types along with the assumed minimum stay times. Figure 2 left shows a one-day trajectory of one test person along with the extracted POI in its surrounding.

POI type	# POI	min. stay time
shop	89,789	10 min.
restaurant	105,665	15 min.
leisure	69,318	15 min.
education	24,151	15 min.
parking	63,602	5 min.
public transport stop	48,463	5 min.
total	**400,988**	–

Table 1: Number of POI extracted from OSM and minimum defined stay time per category

The extraction of visit events is performed by the *local nodes* (see Section 3). A visit results from the spatial intersection of a trajectory and a geographic location and has to last a given minimum period of time. For a formal definition of a visit see [8]. Figure 2 right shows exemplary the extraction of visit events. The POI are colored according to their minimum required stay time (green = 5 minutes, orange = 10 minutes, red = 15 minutes). In the top right picture one visit occurs in the orange colored POI (where a dense cluster of GPS points exists). In the bottom right picture the user passes the POI merely on his way. As the duration of spatial intersection lies below the minimum stay time, no visit events are generated. For the extraction of visits we apply an algorithm from previous work [3], which

[2]http://storm-project.net
[3]http://www.agma-mmc.de
[4]http://www.openstreetmap.org



(a) one-day trajectory of a test person

(b) trajectory excerpts showing one POI visit on top (dense cluster of points) and two POI passages on bottom

Figure 2: left: one-day trajectory of a test person along with OSM points of interest colored according to minimum stay time (green/orange/red = 5/10/15 minutes); top right: visit in POI with 10 minute stay time; bottom right: passages of POI without visiting

# visits per POI	# POI
1	7,590
2	2,176
3	824
4	458
5	223
6	136
7	101
8	53
9	56
≥ 10	192

Table 2: Frequency of visits per POI

was designed to extract visit patterns from a stream of GPS positions online on mobile phones.

In total we extracted 23,508 visit events to 11,809 different POI for all test persons. This number has been considerably below our expectations. Most likely it results from two reasons. First, the number of OSM POI are incomplete. From the online source *http://www.haltestellen-suche.de* we know to expect at least 217,000 stations of public transport in Germany, and also the number of shops in Germany is considerably above the extracted number of POI. Second, GPS signals are typically blocked inside of buildings. As we applied a light-weight event extraction algorithm (that can run on a mobile device), we may have lost a number of visit events.

Table 2 shows an overview of the number of visit events per POI. Most often, only a single visit occurred. This number is quite reasonable given our low number of visits and the independent movement behavior of the test persons.

In order to perform experiments also on a large-scale data set resembling more closely the real-world situation, we replicated the original visit data by a factor of 1,000. We set the time of each such visit by adding Gaussian noise to the current time with $\mu = 0$ and $\sigma = 10{,}000$ seconds.

4.2 Experimental Set-Up

In our experiments we conducted point queries in a sliding window fashion. I.e., we queried the number of events per POI in the past Δt seconds. The selected query windows were of length 30, 600, 1800, 3600 or 86400 seconds. We performed those queries every 10 minutes (in the hierarchical setting this coincides with the time interval of the force action). For our observation period of one week this resulted in $n_t = 1{,}008$ queries per query window for each of the $n_p = 11{,}809$ visited POI. In accordance with our maximum query interval, we set the sliding window parameter of the exponential histogram to 86,400 seconds in all experiments. Further, we varied the maximum acceptable relative error ε to take the values 0.01, 0.02, 0.04, 0.08 and 0.16. In the hierarchical setting we used 10 intermediate nodes which submitted their data structures every 600 seconds to the coordinator.

We measured the error for each experiment using the mean absolute percentage error (MAPE), which is defined as follows:

$$MAPE = \frac{\sum_{i=1}^{n_p} \sum_{j}^{n_t} \left| \frac{x_{ij} - \widehat{x}_{ij}}{x_{ij}} \right|}{n_p \cdot n_t}$$

where x_{ij} denotes the true number of visit events at POI i in query window j and $\widehat{x}_{ij}$ denotes the number of events returned from the exponential histogram. In the case of $x_{ij} = 0$ we added a relative error of zero if our estimate was correct ($\widehat{x}_{ij} = 0$) and a relative error of ∞ if $\widehat{x}_{ij} \neq 0$. This latter case, however, did not occur. We performed all experiments for the flat and hierarchical setting as well as for the original and multiplied data set.

4.3 Results

Figure 3 shows the results for the flat and hierarchical setting of the multiplied data set. The respective numbers are provided in Tables 3 and 4. Note that we display only the results for the multiplied data set because due to the few visit events in the original data set the error was nearly always zero there.

In general, the MAPE is very low, lying with one exception below 1%. For both the flat and hierarchical setting two trends can be observed. First, the MAPE decreases with smaller ε. Second, the MAPE decreases with decreasing size of the query window. The first effect is nearly linear for all query windows and can be expected from the characteristics of exponential histograms. The second is also expected because the error guarantees are given on the size of the sliding window, which was fixed to 86,400 seconds. Accordingly, the error for smaller time intervals has to be lower. However, the effect is linear to the logarithm of the query window sizes, i.e. when increasing the query window, the MAPE increases sublinearly.

When comparing the error between the flat and hierarchical setting, the merge operations result in only a small increase in error.

query wind.	ε=0.01	ε=0.02	ε=0.04	ε=0.08	ε=0.16
30 s	7E-6%	2E-5%	5E-5%	2E-4%	3E-3%
600 s	2E-4%	3E-3%	0.03%	0.18%	0.44%
1800 s	3E-3%	0.04%	0.12%	0.28%	0.54%
3600 s	0.02%	0.06%	0.15%	0.32%	0.59%
86400 s	0.06%	0.14%	0.28%	0.44%	0.79%
mem.	14.5 MB	8.8 MB	5.5 MB	3.4 MB	2.5 MB

Table 3: Mean absolute percentage error and memory usage for flat setting

query wind.	ε=0.01	ε=0.02	ε=0.04	ε=0.08	ε=0.16
30 s	8E-6%	2E-5%	6E-5%	2E-4%	3E-3%
600 s	2E-3%	3E-3%	0.03%	0.18%	0.45%
1800 s	3E-3%	0.04%	0.12%	0.28%	0.64%
3600 s	0.02%	0.06%	0.15%	0.36%	0.75%
86400 s	0.07%	0.16%	0.32%	0.53%	1.03%
mem.	14.5 MB	8.8 MB	5.5 MB	3.4 MB	2.5 MB

Table 4: Mean absolute percentage error and memory usage for hierarchical setting

In order to set the MAPE in perspective to the number of visit events, Table 5 shows the average and maximum number of visits per POI and query interval. The average is hereby calculated once for all POI and time slots and once only for those containing at least one event.

The memory usage of the exponential histogram at the end of the observation period is depicted in the last line in Tables 3 and 4. Assuming fixed 32-bit counters, it depends only on ε and the maximum possible count N in the sliding window of each POI, requiring $O(\frac{1}{\varepsilon}\log N)$ space [1]. As we maintain an exponential histogram for each POI, the required memory depends also linearly on the number of (distinct) visited POI which is, however, constant in our experiments.

query wind.	avg. events	avg. events > 0	max. events
30 s	0.1	1.6	1,916
600 s	2.0	12.1	2,029
1,800 s	5.9	32.0	2,260
3,600 s	11.8	60.0	3,118
86,400 s	270.3	709.8	36,037

Table 5: Number of average and maximum events per POI and query window in ground truth

4.4 Discussion

Our experiments show that the resultant error is very low. For all settings of ε the mean error (MAPE) is less than 1/10 of the maximum acceptable error. This is a very good result. Especially we can be sure for small total number of visits that the query results are always correct. For example, setting $\varepsilon = 0.01$ will result in no errors if less than 100 events occur per POI. This is an important characteristic because the visit frequency of POI is right-tailed, containing only few POI with very high frequencies.

Further the experiments show that our setting scales horizontally. By introducing a layer of 10 intermediate nodes, the MAPE was on average 7.5% higher and at most 23% higher than in the flat setting. Both numbers are considerably below the maximum acceptable error as well as the maximum relative error guaranteed for the join of exponential histograms.

Finally, to evaluate the memory usage, we can compare the numbers to the following baseline scenario. Whenever a visit event occurs, the POI identifier and timestamp are stored at the coordinator using two 4 Byte integers. As our sliding window covers only one day, we will assume that we have to store 1/7 of the total visit events. For the original 23,509 events this results in 0.026 MB. For the multiplied data set it results in 25.6 MB. The storage amount for the original events using exponential histograms varied between 0.54-4.7 MB. In this case we did not save on memory. However, using the more realistic multiplied data set with exponential histogram sizes between 2.5-14.5 MB, our experiments require only 9.7-56.6% of the baseline storage space depending on the selected ε.

When extrapolating to the envisioned setting of monitoring 20 million persons generating each 210 events per day on about 6,500,000 distinct POIs in Germany (including the 6,000,000 distinct street segments), just storing the raw event data would result in 31.3 GB memory consumption. This is considerably above the 1.3-7.8 GB required by the exponential histograms (by just taking into account that our memory consumption increases linearly with the number of distinct POIs).

Considering our entire approach including exponential histograms and local evaluation, the storage reduction is even much higher compared to a naive centralized setting where the users submit a GPS position every second to some central coordinator.

Also note that inserting into and querying an exponential histogram almost takes constant time far below a microsecond, which is much faster than searching an event database of raw events.

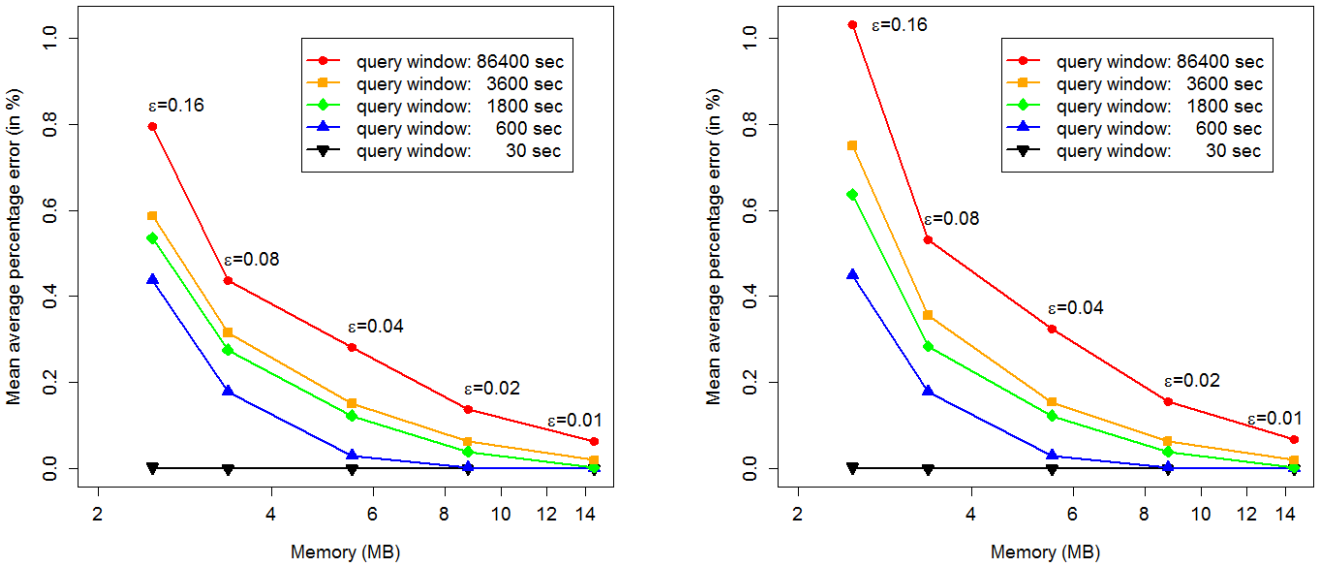


Figure 3: Mean average percentage error and memory usage for different maximum relative errors (ε) and query window sizes; left: without intermediate nodes; right: hierarchy with 10 intermediate nodes

5. CONCLUSIONS

In this paper we propose a distributed system for the large-scale online monitoring of poster performance indicators based on the evaluation of mobility data. Our system relies on the collection and local processing of mobility data via smartphones and uses exponential histograms for the efficient storage and querying of visit statistics in a sliding window fashion. Our experiments on a multiplied real-world data set with nearly 3,000 persons show that the usage of exponential histograms results in an average error of less than 1/10 of the maximum acceptable error while reducing the storage space to an amount as small as 9.7% of the baseline storage space.

6. ACKNOWLEDGMENTS

We thank our colleague Sebastian Bothe for supporting us to run the cluster-based version of the experiments and the Arbeitsgemeinschaft Media-Analyse e.V. for granting the use of the GPS data set. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 255951 (LIFT).

7. REFERENCES

[1] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[2] Fachverband Außenwerbung e.V. Netto-Werbeeinnahmen erfassbarer Werbeträger in Deutschland, 2002-2010 (Net turnover of confirmable advertising media in Gemany, 2000-2010), 2011. `http://www.faw-ev.de/media/download/marktdaten/4\_Nettoumsaetze\_aller_Werbemedien\_ab\_2002.pdf`.

[3] S. Florescu, C. Körner, M. Mock, and M. May. Efficient mobility pattern stream matching on mobile devices. In *Proc. of the Ubiquitous Data Mining Workshop (UDM 2012)*, pages 23–27, 2012.

[4] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Comm. of the ACM*, 42:39–41, 1999.

[5] M. M. Haklay and P. Weber. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[6] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Proc. of the 6th Int. Conf. on Mobile Systems, Applications, and Services (MobiSys'08)*, pages 15–28. ACM, 2008.

[7] C. Kopp, M. Mock, and M. May. Privacy-preserving distributed monitoring of visit quantities. In *SIGSPATIAL 2012 Int. Conf. on Advances in Geographic Information Systems(SIGSPATIAL/GIS)*, pages 438–441, 2012.

[8] C. Körner. *Modeling Visit Potential of Geographic Locations Based on Mobility Data*. PhD thesis, University of Bonn, 2012.

[9] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Greece*, 2011.

[10] Media-Micro-Census GmbH. ma 2012 Plakat - Methoden-Steckbrief zur Berichterstattung, 2012. `http://www.agma-mmc.de/publikationen/methodische-berichte/methoden-steckbriefe.html?eID=dam\_frontend\_push\&docID=179Z`.

[11] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE Int. Conf. on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[12] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *PVLDB*, 5(10):992–1003, 2012.

Multi-Stage Malicious Click Detection on Large Scale Web Advertising Data

Leyi Song, Xueqing Gong*, Xiaofeng He, Rong Zhang, Aoying Zhou
Center for Cloud Computing and Big Data
East China Normal University
3663 North Zhongshan Road, Shanghai, China
songleyi@ecnu.cn, {xqgong,xfhe,rzhang,ayzhou}@sei.ecnu.cn

ABSTRACT

The healthy development of the Internet largely depends on the online advertisement which provides the financial support to the Internet. Click fraud, however, poses serious threat to the Internet ecosystem. It not only brings harm to the advertisers, but also damages the mutual trust between advertiser and ad agency. Click fraud prediction is a typical big data application in that we normally need to identify the malicious clicks from massive click logs, therefore efficient detection methods in big data framework are much desired to combat this fraudulent behavior. In this paper, we propose a three-stage filtering system to attack click fraud. The serialized filters effectively detect the malicious clicks with decreasing confidence that can satisfy both advertisers and content providers.

1. INTRODUCTION

The fast development of the Internet depends not only on the increase of rich content, but also on online advertisement which provides the financial support to the Internet ecosystem. Online advertising tends to benefit all involved parties including content provider, advertiser, ad agency and ad network. It requires the mutual trust among all parties. This trust was at risk, however, by fraudulent clicks. Click fraud (also called click spam, malicious click) is an action of intentional clicking with the purpose of making undue profit, or doing harm to competitors. Click fraud is becoming a serious problem to the World Wide Web [5]. Failing to deter such behaviors will discourage advertisers from actively engaging in more online advertising activities, resulting in less revenue for content providers or ad agencies, and ultimately endangering the Internet ecosystem as a whole. Therefore it is of high importance to detect the malicious clicks.

Ad agencies or networks often deploy different filters to identify malicious clicks [10]. By setting a proper threshold or training a classifier, these methods can handle certain types of anomaly click behaviors generated by human or bots [7, 12, 13]. Despite such efforts, there still remains great challenge to address the problem of malicious clicks. In this paper, we focus on detecting malicious clicks from large scale web advertising data on ad agency side. Ad agency plays an intermediate role between advertisers and publishers in the online advertising system. In order to identify malicious clicks of different categories, for instance, suspicious users, stealthy click-bots and cheating publishers, we build a series of filters at different stages in big data computation framework. The filters are ordered by the decreasing confidence of predicting the malicious clicks. At first stage, a rule based filter identifies the malicious clicks with high confidence since these rules catch strong signals for abnormal clicks. At second stage, a supervised classification approach is used to detect malicious clicks, whose prediction results are of lower confidence than rule based ones. Finally we cluster the clicks into group at stage 3 with the hope that fraudulent clicks generated from one publisher will be grouped together. The clustering method is an unsupervised learning process, hence results in lowest prediction confidence. Sequentially organizing the filters in decreasing confidence order provides us with flexibility and scalability to add extra filters. For instance, we can replace one classifier with another classifier at stage 2, or add more classifiers to stage 2 to form an ensemble. Users have the freedom to use the result corresponding to different confidence.

We use the precision as the measure of confidence in this paper. The reason to use precision, instead of other metrics such as recall or F-measure is because 1) precision focuses on singling out bad clicks while trying to minimize the possibility of predicting valid clicks as fraudulent ones; 2) it is well-known that the judgement of whether or not a click is a fraudulent one is very subjective in many cases. Too large recall can potentially classify large number of valid clicks as bad ones unless we have high quality classifiers which is hard to obtain, especially when human judgement is difficult.

In this paper, we propose a click fraud detecting architecture which organizes filters sequentially by decreasing confidence order. This architecture was applied to large scale advertising log obtained from an ad agency. Specifically,

- we address the malicious click problem on the ad agency side. In addition to rule-based and supervised methods, we propose a clustering-based method to analyse the traffic quality;
- we design a click detecting mechanism in big data computation framework, i.e., Hadoop, to detect malicious

* Corresponding Author

clicks efficiently by sequentially organizing three types of filters of different confidence level;

- we carefully design and analyse important features, and verify our approach by evaluating the dissimilarity between predicted fraudulent clicks *vs.* valid clicks.

The rest of the paper is organized as follows. In Section 2, we introduce previous work related to click fraud detection. We present our detection architecture in Section 3. In Section 4, we comprehensively analyze the results of our approach by applying it to over one month real click log data from an ad agency, and conclude our paper in Section 5.

2. RELATED WORK

There are many participants play in online advertising ecosystem. Ad agency usually plays an match maker role between publishers and advertisers. Publishers own the website pages containing ad slots. Advertisers purpose their ad creativity to attract users to buy their products or to make other kinds of profit. Ad agency buys inventory from publishers and sells advertising traffic to advertisers. Thus, for a reputed ad agency, it is responsible to filter these malicious clicks before charging.

Undoubtedly there are great efforts in the area of fraud detection, including the click fraud problem, but most solutions are not easily available. The report by Tuzhilin [10] introduced some of the approaches that Google adapted to fight click fraud, in which both rule-based and anomaly-based filters were incorporated into search engine. Since search advertising is one of the major forms of online advertisement, this work inspired us to take similar approach in fraud detection system.

Another type of malicious click detection methods was based on click stream analysis techniques which identified patterns of fraudulent traffic. Metwally *et al.* proposed fraud detection solutions for data stream by combining association rules and duplicate detection methods [8]. Effective data structure such as modified Bloom Filter was used in this situation [13]. However, it is often difficult for most data mining-based detection methods to be implemented into stream analysis, hence limits the adaptation of such techniques to click fraud detection.

Supervised learning approach detects the malicious clicks by training a classifier. Data collection is one of the most important steps in this approach. Haddadi [3] proposed the idea of bluff ads, which is unrelated to the user search, user profile and recently activity. If a high ratio of such ads was clicked, the user could be flagged as suspicious. In some other work, CAPTCHA was used for training data generation and useful data collection [6]. For ad agencies, they have various advertiser and publisher sources, hence CAPTCHA approach is hardly applicable. The next key step is to extract features. The work [1] investigated query attributes between human and robot traffic. Different type of features could be extracted from the click attributes and user attributes, such as click count or geographic origin [1, 4].

Bot-generated click traffic is a big part of malicious clicks. The state-of-the-art bot detection work mostly aimed at clicks in search engine logs [6, 12]. Yu *et al.* proposed SBot-Miner, a system which automatically identified bot generated search traffic from query log, using history-based as well as matrix-based unsupervised methods [12].

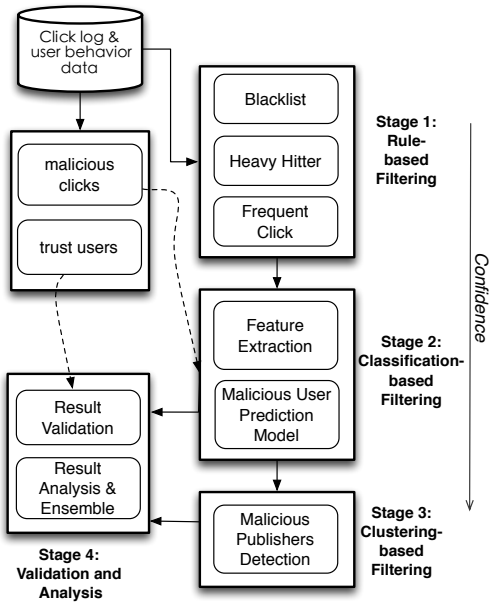


Figure 1: Architecture Overview.

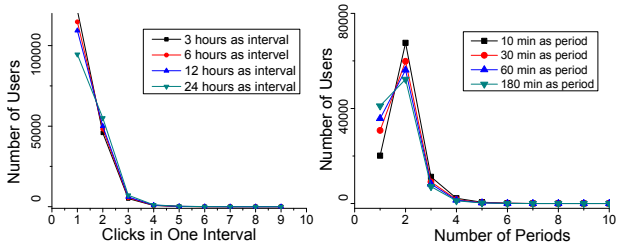
Previous approaches to identify and understand malicious clicks focused on one specific method, or one specific problem. Researchers adopted click-through rate related methods for web spam in search engine [11]. However, it is difficult for an ad agency to access the click-though rate, therefore some fraudulent patterns can not be easily identified by traditional methods. In our framework, we design a more general way of data collection and filtering strategy for malicious click detection, implemented in a stage-wise architecture.

3. STAGE-WISE CLICK FRAUD FILTERING ARCHITECTURE

3.1 Architecture Overview

In order to accommodate different filtering methods with different confidence in predicting the fraudulent clicks, we argue that it is advantageous to take the approach of stage-wise filtering architecture. The filters are sequentially connected such that the filter generating results with higher predicting confidence is put in the chain before the one with lower confidence. The intuition is that we need to identify most confident malicious clicks, and reduce the data size which need further processing, hence reduce the complexity of the problem. Furthermore, stage-wise structure offers prediction results of different confidence, which enables the users to utilize the result with more flexibility.

The stage-wise filtering system architecture is illustrated in Figure 1. The major components are following 3 stages. 1) Rule-based filters at stage 1 to detect obvious invalid clicks with high confidence. They can identify two types of malicious clicks: heavy hitter and frequent clicker. 2) Classification-based filters at stage 2 to determine more complicated clicks with human judged training set. 3) The clustering-based filter at stage 3 to identify cheating groups from similar publisher websites. We use intra-cluster distance and query diversity to separate malicious groups.



(a) Click density (b) Click frequency

Figure 2: Click statistics of a trusted user click log dataset in one month

3.2 Rule-based Filtering

To fight click fraud, generating blacklist in the filtering system is the most reliable method. It is easy to compile the blacklist for violating entities such as user agent(UA) or IP address, but the coverage of a blacklist is limited. Setting certain rules is efficient to exclude more malicious clicks from entering accounting system. In this paper we focus on setting rules for the heavy hitter and frequent click problems [8, 13, 7].

Heavy hitter in click logs means, at a specific time interval, the click rate of a user is relatively higher than a threshold λ_1, while frequent click problem refers to the situation that user's click appears in relatively more periods[1] than a predefined threshold λ_2.

To obtain a reasonable filtering threshold, we note from Figure 2 that the number of clicks and periods follows the *Zipfian distribution*. We set the maximum number for normal user behavior as the lower value of the two: number of clicks in one interval and number of periods the user clicks. For better accuracy, we can also determine the threshold based on the p-quantile value on the entire log dataset.

3.3 Classification-based Filtering

Classification-based methods are widely used in fraud detection or spam detection field [9]. Classification is an effective way for addressing malicious clicks, especially for stealthy clicks that are hard to be captured by rule-based approach. The classification has been applied to real data with success [4, 10]. One of the biggest advantages with classification approach is that once a model has been built, the prediction of new instance is usually quite fast. There is also research on the attributes which can distinguish human and bot traffic [1]. In our work, we use the traditional features used in previous work, and also engineer new features useful for training classifiers. We will discuss the features in more details later. The result of this stage is a set of malicious users from the click log.

3.4 Clustering-based Filtering

It is obvious that results of supervised methods highly depend on the accuracy, coverage and labeling scheme of the labeled corpus. The classified malicious clicks are limited by the fraudulent types in training set. Hence, we develop our clustering-based method, which is based on our observation that, for agency's customers(advertisers), some of the suspicious ad traffic from the same website shows high similarity. We try to group similar traffic and analyze them as a group in order to detect abusive clicks from the publisher-side. This approach can also be applied to search engine traffic, since the query diversity can be used to separate the suspicious groups from the search ad log. Considering the result confidence and filtering cost, this stage can be an optional choice for each advertiser. For valuable customers, this is an attractive feature, which differs from previous work.

For clustering, We define the **dissimilarity** between two log entries x, y as:

$$D(x,y) = \sum_{f \in Fields} w_f d_f(x,y) \tag{1}$$

where each log entry is represented by a vector of click attributes as $< user, IP, referrer, UA, area, query >$. $Fields$ is the feature set used in click attributes and $d_f(x,y)$ is the distance measure defined on each attribute f, normalized between [0,1]. w_f is the weight of $d_f(x,y)$.

For attributes like refer URL and user agent, we care about the longest matching prefix, and the distance measure is defined as:

$$d_{url}(x,y) = 1 - \frac{LCP(x,y)}{\max(|x|,|y|)} \tag{2}$$

where LCP means the longest common prefix of two strings.

Even though network address translation (NAT) might allow many users behind a single IP address, the cheating groups always show strong similarity in IP address. To amplify the importance of the tail part in IP address, we take 32 bits IPv4 address as an example to define the distance. If $LCB(longest\ common\ bits) \geq 16$, then

$$d_{IP}(x,y) = 1 - \frac{LCB(x,y)}{total\ bits}, \tag{3}$$

otherwise, the distance between two IP addresses is 1.

For other attributes such as area, we simply treat their dissimilarity as binary value (0 or 1). Furthermore, we can easily prove that $D(x,y)$ is a metric, since it follows the properties: (1) $D(x,y) = 0 iff x = y$, (2) $D(x,y) \geq 0$, (3) $D(x,y) = D(y,x)$, and (4) $D(x,y) \leq D(x,z) + D(z,y)$(triangle inequality). Due to space limit, we skip the proof here.

After obtaining the dissimilarity matrix of a log set, we use *k-medoids* algorithm to produce the clustering in this stage. K-medoids is an adaptation of k-means algorithm. Rather than calculating the mean of the items for each cluster, which is not applicable in our situation, a representative item, or medoid, is chosen for each cluster. Medoids for each cluster are calculated to finding object i by minimizing

$$\tilde{J} = \sum_{j \in C_i} D(i,j) \tag{4}$$

where C_i is the cluster containing object i and $D(i,j)$ is the dissimilarity function defined in equation 1. Since the algorithm simply looks up the dissimilarity matrix, it only needs to be calculated once in the beginning.

The next step is how to distinguish cheating groups from all clusters. Obviously, if one group agrees on most fields, it indicates this group of clicks come from a botnet using similar terminals and browsers or a real interested user with high probability. From the click statistics we can figure that,

[1]We use *period* to represent the window for counting in frequent clicks problem, in order to distinguish the *interval* of heavy hitter. Clicks of each user occur within the same period will be ignored.

Table 1: Examples of Labeled Malicious Clicks

User	Advertiser	Area	IP	Referer	Query	User agent
1	c1	1	x.x.25.177	none	none	Mozilla/5.0
2	c2	1	x.x.25.178	none	none	Mozilla/5.0
3	c3	2	y.y.51.137	http://r1.com/s?wd=wvihv	wvihv	Mozilla/4.0(compatible; MSIE 7.0;)
3	c3	2	y.y.51.142	http://r1.com/s?wd=jmfitxm	jmfitxm	Mozilla/4.0(compatible; MSIE 7.0;)
4	c3	2	y.y.51.145	http://r2.com/s?wd=qyfsoc	qyfsoc	Mozilla/4.0(compatible; MSIE 7.0;)

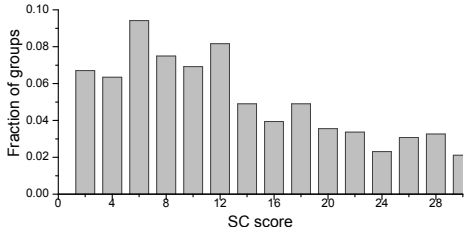


Figure 3: Fraction of groups vs. Scatter scores

if the group size is reasonable, it is less possible to be an interested user. In other words, if the intra-cluster similarity of a group is lower, then the probability of the traffics in the group being malicious is higher. Moreover, the high similarity of the referrer in a suspicious group means the low traffic quality of the publisher except search engine. In particular, we add the query diversity factor for search engine traffic. We define the *Scatter(SC)* score of each group as:

$$SC = intra\_distance \times query\_diversity \quad (5)$$

where *query_diversity* is defined as the ratio of distinct search phrases to total search phrases. The *query_diversity* is set to 1 for non-search engine traffic. By adding query diversity, we want to give more importance to the website traffic groups, since it is more difficult to locate the root cause of problem in search engine groups. Thus, we use the intra-cluster distance times query diversity to measure the ads click *Scatter*-ness of groups. Finally, the groups with low *SC* score will be regarded as suspicious groups. Figure 3 shows the distribution of *Scatter* score across our test groups. A brief description is shown in Algorithm 1.

input : clicks on each advertiser
output: groups of malicious clicks
initialize dissimilarity matrix;
while *not at end of advertisers' traffic set* **do**
  step 1: apply K-medoids using $D(x, y)$ distance to get traffic groups with similar features
  step 2: select suspicious groups with lower SC score
end

Algorithm 1: Major steps of getting cheating groups

3.5 Feature Extraction

The features extracted from data that work in web page spam domains may not work in ad log analysis. We inspect the click data and introduce several features specifically designed for classifiers to predict malicious clicks. Features are created by studying the labeled users' activity patterns.

In order to define useful features, we need to analyze the difference between normal and malicious click behaviors. Below, we discuss some of the features we identified to represent the user.

Number of clicked advertisers. This feature counts the number of distinct advertisers each user clicked. Malicious users show extreme patterns, for instance most have empty cookie and others have dense clicks on one advertiser.

Click ratio on advertisers. This feature takes into account both the total clicks and the distinct clicked advertisers, defined as total clicks/total clicked advertisers. For each user, it represents the average clicks per advertiser. We observe that the trusted users show higher diversity by comparing their histograms.

We also define features that can be used to characterize the attribute of a user. For instance, a fraudulent user might carry out the malicious click behaviors from one device, but with many dynamically allocated IP addresses. Thus, we derive features from user agent, IP and cookie. Short cookies are more suspicious than normal cookies, same goes for user agent. Some details about these features are shown below.

Click/IP ratio. This feature is defined as the total clicks for a user/total unique IP addresses the clicks come from.

Variance of IP clicks. After counting the number of clicks from each IP address, we can calculate the variance of these clicks. It will be suspicious if one user launches clicks with consistent frequency from some IP addresses. Besides, we also use features such as the number of user agents, number of referrers, length of agent, length of cookie.

There are some other features derived from geographic or temporal attributes. For click timestamp, we divide a day into four six-hour periods: night(0:00 to 5:59), morning(6:00 to 11:59), afternoon(12:00 to 17:59) and evening(18:00-23:59). With the information available, the features below are also extracted: *most frequent area*, *most frequent period*, *number of clicks in each period*, *mean/std deviation of clicks in periods* and so on. We are not going to list out all the 17 features used in classification considering the space limit.

4. EXPERIMENTS

In this section we present the experimental results of the stage-wise filtering framework.

4.1 Dataset

The data set used in the experiments is over one month click log we get from an ad agency company. It contains about 35 million records of ad clicks. The advertising log data normally contains attributes as user ID, click timestamp, user's IP, cookie, query phrase (*if the ad traffic is from a search engine*), user agent and referrer (*url of the page where user clicked the ad link*).

To protect privacy, users who click the ads are assumed to be only temporarily identified by cookie. We assume that

Table 2: Brief description of training set.

Description	# of Clicks/Users
Fraudulent clicks	174,642
Non-fraudulent clicks	779,980
Malicious users	83,061
Benign users	649,204

some user actions such as buying stuff or registering are benign. In this way, we extract non-fraudulent clicks from benign users for training. On the other hand, some malicious clicks were addressed using domain knowledge. Examples of malicious clicks are shown in Table 1. We notice that attack may come from similar IP addresses with fake user agent, fake referrer, or meaningless query phrase. Besides, the training data includes complete clicks of three advertisers that can be used to evaluate clustering performance. A brief description of dataset for training is shown in Table 2.

4.2 Experimental Setup

We run our stage-wise method on an 8-node Hadoop cluster using Pig Latin [2]. We split each stage into Map/Reduce modules in pipeline. Each module can be converted into one or several rounds of Map/Reduce tasks. For example the filtering phase in rule-based stage, users are partitioned through mappers and the counting filter UDFs(User Define Functions) on each user are performed in reducer. How to minimize the I/O cost is a major research for analysis on large scale dataset. Taking this problem into consideration, briefly, we design our UDFs to get the most results through one-pass over the data.

In the first stage of our framework, the bound rules for click density and click frequency are determined using p-quantile of the entire distribution. From our training data, we find there contains over 0.5% heavy hitters. There exists a certain percentage of noisy clicks in the log, 0.995 or similar (consider the distributions in Figure 1) can be chosen as value of p in our experiment for rule setting. The large click logs are filtered by passing through click counting, period counting, quantile calculating and filtering phases.

Next, the reduced dataset is passed to feature extraction modules as well as classifiers in the second stage. In our experiment, 17 features which have been discussed above are created. Then, we use training set in Table 2 to train classification models and compare the performance of the following classifiers: Naïve Bayes, Decision Table, Bayes Net, REP Tree and Random Forest(10 trees, 5 random features each). The comparison results of 5-fold cross validation over training set are presented in Table 3. Since the labelled malicious are just partial of real malicious clicks, the relatively recall performance is less important than precision. The performance could be very good result of the simple labeling scheme, thus we need unsupervised methods.

Finally, three advertisers with different click size(100K, 10K, 1K) are chosen as the evaluation dataset in the clustering stage. We set equal weight for different features in experiment. According to Figure 3, we use 2.0 as the lower bound for the SC score of groups, which means the group is quite dense. The number of K in applying K-medoids are determined experimentally.

4.3 Results Analysis and Validation

Table 3: Precision performance for classifiers.

Classifier	% Precision	Run Time (s)
Random Forest	**97.6**	322
REP Tree	**96.8**	98
Bayes Net	96.7	115
Decision Table	96.1	199.5
Naïve Bayes	52.9	27

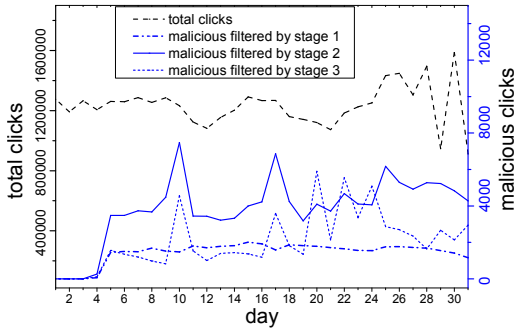


Figure 4: Analysis of the click log dataset.

Figure 4 illustrates the result of our implementation of stage-wise approach applied on click log of one month.

From the daily result, we find that a high percentage of clicks follows the format *http://search-engine.com/s?word* with the same short UA. Further study suggests that these clicks are coming from bot-net which directly inject noise into our logs using fake referer, user agent and IP address. We also find four suspicious publisher groups, which generate high density clicks with similar IP addresses and user agents from their website. Moreover, part of the suspicious clicks were generated by download manager showing in UA.

Figure 4 shows that the percentage of malicious in first stage distributed evenly. We could reasonably assume that all the heavy hitters and frequent clicks are malicious, since the upper bounds were setting based on the propriety of distribution. Thus, the rule-based filter can be chosen as the basic filter with absolute high confidence to meet basic requirement of ad agency. For classification stage, we choose Random Forest as our model. This stage can discover stealthy clicks with suspicious patterns derived from domain knowledge-based labeling. However, for the limitation of supervised method, false positive is inevitable. Our aim is to get the classifier with maximum precision. As to the clustering stage, we evaluate the precision on three different advertisers' click log. If the number of final result groups is n, precision metric in this case is:

$$Precision = \frac{1}{n}\sum_{i}^{n}\frac{|C_i \cap L|}{|C_i|}$$

where L is the cheating click set in result, and C_i is the total click set from each of the advertisers. Figure 5 shows our test results of clustering. Three advertisers' dataset achieve different precision performance, which indicates that the confidence partially depends on the distribution of real click data. Therefore, we set the clustering-based filters in the last stage. Indeed, even this filter has a lower confidence, it

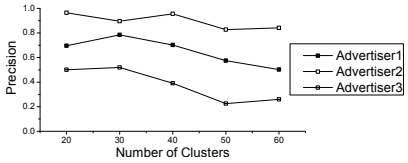


Figure 5: Precision performance of clustering method on evaluation data from three advertisers.

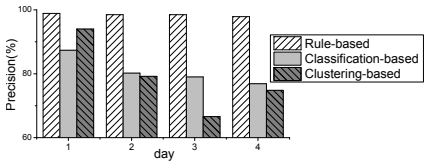


Figure 6: Precision performance of three stages on sampled result.

is important for ad agency to assess the traffic quality from publishers by evaluating the malicious group.

It is worth pointing out that our framework built on the top of Hadoop platform achieved high efficiency for processing click log. Due to the as-many-as computation in one-pass design, each stage could be finished in minutes for both daily and monthly filtering.

To further validate the stage-wise precision, we pick 4-day predicated malicious clicks for human judgement. We invite domain expert to inspect these clicks and the comparison results were shown in Figure 6. We see that the rule-based methods achieve high precision, which verifies our assumption. However, false positives are inevitable in any unsupervised learning algorithms. It is interesting that most of the false classified results are from mobile applications, especially on the 10th day. The missing of referrer field in these clicks is the root cause, which makes the patterns of these clicks similar to those of malicious clicks.

As a comparison, we check the diversity ratio on three features: hash of cookie, hash of user agent, hash of referrer, by sampling clicks from positive and negative results respectively. The diversity is defined as the ratio of distinct items to total samples. Figure 7 shows the result: diversities of normal clicks are relatively higher than these of malicious groups. For example, a group of intentional clicks with similar UA may come from one commander. The obvious difference between the malicious groups and normal groups suggests that the identified ones are indeed very suspicious.

5. CONCLUSION

Fraudulent click is a malicious behavior which threatens the healthy development of Internet ecosystem. In this work, we propose a stage-wise click fraud filtering architecture which effectively identifies the fraud clicks for ad agency with different prediction confidence. The stages in this work can be further divided into a set of modules, which consist of one or several rounds of Map/Reduce using parallel computing. We performed an in-depth analysis on one month click log using the proposed framework and evaluated our results by different metrics.

6. ACKNOWLEDGMENTS

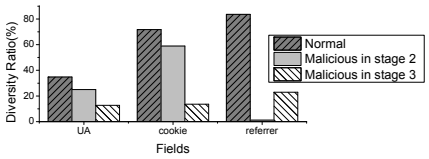


Figure 7: Diversity comparison on three features.

This work is partially supported by the Key Program of National Natural Science Foundation of China grant No. 61232002, National Science Foundation of China under grant No.60925008, No.61103039, No.61021004 and the Key lab Project of Wuhan University.

7. REFERENCES

[1] O. Duskin and D. G. Feitelson. Distinguishing humans from robots in web search logs: preliminary results using query rates and intervals. In *Proceedings of the 2009 workshop on Web Search Click Data*, WSCD '09, pages 15–19, 2009.

[2] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2), 2009.

[3] H. Haddadi. Fighting online click-fraud using bluff ads. *Computer Communication Review*, 40(2):21–25, 2010.

[4] M. Hager and T. Landergren. Implementing best practices for fraud detection on an online advertising platform. Master's thesis, Chalmers University of Technology, 2010.

[5] B. J. Jansen. Click Fraud. *Computer*, 40(7):85–86, July 2007.

[6] H. Kang, K. Wang, D. Soukal, F. Behr, and Z. Zheng. Large-scale bot detection for search engines. In *WWW*, pages 501–510, 2010.

[7] B. Lahiri, J. Chandrashekar, and S. Tirthapura. Space-efficient tracking of persistent items in a massive data stream. In *DEBS*, pages 255–266, 2011.

[8] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.

[9] C. Phua, D. Alahakoon, and V. C. S. Lee. Minority report in fraud detection: classification of skewed data. *SIGKDD Explorations*, 6(1):50–59, 2004.

[10] A. Tuzhilin. The lane's gifts v. google report. *http : //googleblog.blogspot.in/pdf/Tuzhilin$_R$eport.pdf*, 2007.

[11] C. Wei, Y. Liu, M. Zhang, S. Ma, L. Ru, and K. Zhang. Fighting against web spam: a novel propagation method based on click-through data. In *SIGIR*, pages 395–404, 2012.

[12] F. Yu, Y. Xie, and Q. Ke. Sbotminer: large scale search bot detection. In *WSDM*, pages 421–430, 2010.

[13] L. Zhang and Y. Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *ICDCS*, pages 77–84, 2008.