Incremental Sliding Window Analytics

Pramod Bhatotia and Umut A. Acar and Flavio P. Junqueira and Rodrigo Rodrigues

Abstract Sliding-window computations are widely used for large-scale data analysis, particularly in live systems where new data arrives continuously. These computations consume significant computational resources because they usually recompute over the full window of data every time the window slides. In this chapter, we propose techniques for improving the scalability of sliding-window computations by performing them incrementally. In our approach, when some new data is added at the end of the window or old data dropped from its beginning, the output is updated automatically and efficiently by reusing previously run sub-computations. The key idea behind our approach is to organize the sub-computations as a shallow (logarithmic depth) balanced tree and perform incremental updates by propagating changes through this tree. This approach is motivated and inspired by advances on self-adjusting computation, which enables automatic and efficient incremental computation. We present an Hadoop based implementation that also provides a dataflow query processing interface. We evaluate it with a variety of applications and real-world case studies. Our results show significant performance improvements for large-scale sliding-window computations without any modifications to the existing data analysis code.

Pramod Bhatotia
University of Edinburgh, e-mail: pramod.bhatotia@ed.ac.uk

Umut A. Acar
CMU, e-mail: umut@cs.cmu.edu

Flavio P. Junqueira
Dell EMC, e-mail: fpj@apache.org

Rodrigo Rodrigues
IST Lisbon, e-mail: rodrigo.miragaia.rodrigues@tecnico.ulisboa.pt

1 Introduction

There is a growing need to analyze large feeds of data that are continuously collected. Either due to the nature of the analysis, or in order to bound the computational complexity of analyzing a monotonically growing data set, this processing often resorts to a *sliding window* analysis. In this type of processing, the scope of the analysis is limited to a recent interval over the entire collected data, and, periodically, newly produced inputs are appended to the window and older inputs are discarded from it as they become less relevant to the analysis.

The basic approach to sliding-window data processing is to recompute the analysis over the entire window whenever the window slides. Consequently, even old, unchanged data items that remain in the window are reprocessed, thus consuming unnecessary computational resources and limiting the timeliness of results.

We can improve on this using an incremental approach, which normally relies on the programmers of the data analysis to devise an incremental update mechanism [25, 28, 33], i.e., an *incremental algorithm* (or *dynamic algorithm*) containing the logic for incrementally updating the output as the input changes. Research in the algorithms and programming languages communities shows that while such incremental algorithms can be very efficient, they can be difficult to design, analyze, and implement even for otherwise simple problems [1, 18, 22, 38]. Moreover, such incremental algorithms often assume a uniprocessor computing model, and due to their natural complexity do not lend themselves well to parallelization, making them ill-suited for parallel and distributed data analysis frameworks [21, 26].

Given the efficiency benefits of incremental computation, an interesting question is whether it would be possible to achieve these benefits without requiring the design and implementation of incremental algorithms on an ad hoc basis. Previous work on systems like Incoop [12, 13], Nectar [24], HaLoop [14], DryadInc [26], or Ciel [31] shows that such gains are possible to obtain in a transparent way, i.e., without changing the original (single pass) data analysis code. However, these systems resort to the memoization of sub-computations from previous runs, which still requires time proportional to the size of the whole data rather than the change itself. Furthermore, these systems are meant to support arbitrary changes to the input, and as such do not take advantage of the predictability of changes in sliding window computations to improve the timeliness of the results.

In this paper we propose SLIDER, a system for incremental sliding window computations where the work performed by incremental updates is proportional to the size of the changes in the window (the "delta") rather than the whole data. In SLIDER, the programmer expresses the computation corresponding to the analysis using either MapReduce [21] or another dataflow language that can be translated to the MapReduce paradigm (e.g., Hive [23] or Pig [15]). This computation is expressed by assuming a static, unchanging input window. The system then guarantees the automatic and efficient update of the output as the window slides. The system makes no restrictions on how the window slides, allowing it to shrink on one end and to grow on the other end arbitrarily (though as we show more restricted changes lead

to simpler algorithms and more efficient updates). SLIDER thus offers the benefits of incremental computation in a fully transparent way.

Our approach to automatic incrementalization is based on the principles of self-adjusting computation [1,2], a technique from the programming-languages community that we apply to sliding window computations. In self-adjusting computation, a *dynamic dependency graph* records the control and data dependencies of the computation, so that a *change-propagation algorithm* can update the graph as well as the data whenever the data changes. One key contribution of this paper is a set of novel data structures to represent the dependency graph of sliding window computations that allow for performing work proportional primarily to the size of the slide, incurring only a logarithmic—rather than linear—dependency to the size of window. To the best of our knowledge, this is the first technique for updating distributed incremental computations that achieves efficiency of this kind.

We further improve the proposed approach by designing a *split processing model* that takes advantage of structural properties of sliding window computations to improve response times. Under this model, we split the application processing into two parts: a foreground and a background processing. The foreground processing takes place right after the update to the computation window, and minimizes the processing time by combining new data with a pre-computed intermediate result. The background processing takes place after the result is produced and returned, paving the way for an efficient foreground processing by pre-computing the intermediate result that will be used in the next incremental update.

We implemented SLIDER by extending Hadoop and evaluate its effectiveness by applying it to a variety of micro-benchmarks and applications and consider three real-world cases [8, 9]. Our experiments show that SLIDER can deliver significant performance gains, while incurring only modest overheads for the initial run (non-incremental pre-processing run).

2 Background and Overview

In this section, we present some background and an overview of the design of SLIDER.

2.1 Self-Adjusting Computation

Self-adjusting computations is a field that studies ways to incrementalize programs automatically, without requiring significant changes to the code base [1,6]. For automatic incrementalization, in self-adjusting computations, the system constructs and maintains a *dynamic dependency graph* that contains the input data to a program, all sub-computations, and the data and control dependencies in between, e.g., which outputs of sub-computations are used as inputs to other sub-computations, which sub-computations are created by another. The dynamic dependency graph enables

a *change propagation* algorithm to update the computation and the output by propagating changes through the dependency graph, re-executing all sub-computations transitively affected by the change, re-using unaffected computations, and deleting obsolete sub-computations which no longer take place. The change propagation algorithm has been shown to be effective for a broad class of computations called *stable* and has even helped solve algorithmically sophisticated problems in a more efficient way than more ad hoc approaches (e.g., [3, 39].)

2.2 Design Overview

Our primary goal is to design techniques for automatically incrementalizing sliding-window computation to efficiently update their outputs when the window slides. We wish to achieve this without requiring the programmer to change any of the code, which is written assuming that the windows do not slide, i.e., the data remains static. To this end, we apply the principles of self-adjusting computation to develop a way of processing data that leads to stable computation, i.e., one whose overall dependency structure does not change dramatically when the window slides. In this paper, we apply this methodology to the MapReduce model [21], which enables expressive, fault-tolerant computations on large-scale data, thereby showing that the approach can be practical in modern data analysis systems. Nonetheless, the techniques we propose can be applied more generally, since we only require that the computation can be divided in a parallel, recursive fashion. In fact, the techniques that we propose can be applied to other large-scale data analysis models such as Dryad [26], Spark [43], Pregel [30], which allow for recursively breaking up a computation into sub-computations.

Assuming that the reader is familiar with the MapReduce model, we first outline a basic design and then identify the limitations of this basic approach and describe how we overcome them. For simplicity, we assume that each job includes a single Reduce task (i.e., all mappers output tuples with the same key). By symmetry, this assumption causes no loss of generality; in fact our techniques and implementation apply to multiple keys and reducers.

The basic design. Our basic design corresponds roughly to the scheme proposed in prior work, Incoop [12, 13], where new data items are appended at the end of the previous window and old data items are dropped from the beginning. To update the output incrementally, we launch a Map task for each new "split" (a partition of the input that is handled by a single Map task) that holds new data and reuse the results of Map tasks operating on old but live data. We then feed the newly computed results together with the re-used results to the Reduce task to compute the final output.

Contraction trees and change propagation. The basic design suffers from an important limitation: it cannot reuse any work of the Reduce task. This is because the Reduce task takes as input all values for a given key ($< K_i >$, $< V_1, V_2, V_3, ...V_n >$), and therefore a single change triggers a re-computation of the entire Reduce task. We address this by organizing the Reduce tasks as a contraction tree and proposing

a change-propagation algorithm that can update the result by performing traversals on the contraction tree, while maintaining it balanced (low-depth) and thus guaranteeing efficiency.

Contraction trees (used in Incoop [13], Camdoop [20], iMR [29]) leverage *Combiner functions* of MapReduce to break a Reduce task into smaller sub-computations that can be performed in parallel. Combiner functions originally aim at saving bandwidth by offloading parts of the computation performed by the Reduce task to the Map task. To use combiners, the programmer specifies a separate Combiner function, which is executed on the machine that runs the Map task, and performs part of the work done by the Reduce task in order to pre-process various <key,value> pairs, merging them into a smaller number of pairs. The combiner function takes both as an input and output type a sequence of <key,value> pairs. This new usage of Combiners is compatible with the original Combiner interface, but requires associativity for grouping different Map outputs. To construct a contraction tree, we break up the work done by the (potentially large) Reduce task into many applications of the Combiner functions. More specifically, we split the Reduce input into small groups, and apply the Combine to pairs of groups recursively in the form of a balanced tree until we have a single group left. We apply the Reduce function to the output of the last Combiner.

When the window slides, we employ a *change-propagation algorithm* to update the result. Change propagation runs Map on the newly introduced data and propagates the results of the Map tasks through the contraction tree that replaced the Reduce task. One key design challenge is how to organize the contraction tree so that this propagation requires minimal amount of time. As explained in Section **??**, we overcome this challenge by designing contraction trees and the change-propagation algorithm such that change propagation always guarantees that the contraction trees remain balanced, guaranteeing low depth. As a result, change propagation can perform efficient updates by traversing only through a short path of the contraction tree. In this approach, performing an update not only updates the data but also the structure of the contraction tree (so that it remains balanced), guaranteeing that updates remain efficient regardless of the past history of window slides.

In addition to contraction trees and change propagation, we also present a *split processing* technique, that takes advantage of the structural properties of sliding-window computation to improve response times. Split processing splits the total work into background tasks, which can be performed when no queries are executing, and foreground tasks which must be performed in order to respond to a query correctly. By pre-processing certain computation in the background, split processing can improve performance significantly (up to 40% in our experiments).

The techniques that we present here are fully general: they apply to sliding window computations where the window may be re-sized arbitrarily by adding as much new data at the end and removing as much new data from the beginning as desired. As we describe, however, more restricted updates where for example the window is only expanded by append operations (*append-only*), or where the size of the window remains the same (*fixed-size windows*), lead to a more efficient and simpler design, algorithms, and implementation.

3 Slider Architecture

In this section, we give an overview of our implementation. Its key components are described in the following.

Implementation of self-adjusting trees. We implemented our prototype of SLIDER based on Hadoop. We implemented the three variants of the self-adjusting contraction tree by inserting an additional stage between the shuffle stage (where the outputs of Map tasks are routed to the appropriate Reduce task) and the sort stage (where the inputs to the Reduce task are sorted). To prevent unnecessary data movement in the cluster, the new contraction phase runs on the same machine as the Reduce task that will subsequently process the data.

SLIDER maintains a distributed dependency graph from one run to the next and pushes the changes through the graph by re-computing sub-computations affected by the changed input. For this purpose, we rely on a memoization layer to remember the inputs and outputs of various tasks and the nodes of the self-adjusting contraction trees. A shim I/O layer provides access the memoization layer. This layer stores data in an in-memory distributed data cache, which provides both low-latency access and fault tolerance.

Split processing knob. SLIDER implements the split processing capability as an optional step that is run offline on a best effort basis. This stage is scheduled during low cluster utilization periods as a background task. It runs as a distributed data processing job (similar to MapReduce) with only one offline pre-processing stage.

In-memory distributed data caching. To provide fast access to memoized results, we designed an in-memory distributed data caching layer. Our use of in-memory caching is motivated by two observations: First, the number of sub-computations that need to be memoized is limited by the size of the sliding window. Second, main memory is generally underutilized in data-centric computing [4]. The distributed in-memory cache is coordinated by a master node, which maintains an index to locate the memoized results. The master implements a simple cache replacement policy, which can be changed according to the workload characteristics. The default policy is Least Recently Used (LRU).

Fault-tolerance. Storing memoized results in an in-memory cache is beneficial for performance, but can lead to reduced cache effectiveness when machines fail, as it requires unnecessary re-computation (especially for long running jobs). We therefore conduct a background replication of memoized results to provide fault tolerance by creating two replicas of each memoized result (similar to RAMCloud [32]). This way fault tolerance is handled transparently: when a new task wants to fetch a memoized result it reads that result from one of the replicas. To ensure that the storage requirements remain bounded, we developed a garbage collection algorithm that frees the storage used by results that fall out of the current window.

Scheduler modifications. The implementation of SLIDER modifies Hadoop's scheduler to become aware of the location of memoized results. Hadoop's scheduler chooses any available node to run a pending Reduce task, only taking locality into account when scheduling Map tasks by biasing towards the node holding the input.

SLIDER's scheduler adapts previous work in data-locality scheduling [13,40–42], by attempting to run Reduce tasks on the machine that contains the memoized outputs of the combiner phase. When that target machine is overloaded, it migrates tasks from the overloaded node to another node, including the relevant memoized results. Task migration is important to prevent a significant performance degradation due to straggler tasks [44].

Dataflow query processing. As SLIDER is based on Hadoop, computations can be implemented using the MapReduce programming model. While MapReduce is gaining in popularity, many programmers are more familiar with query interfaces, as provided by SQL or LINQ. To ease the adoption of SLIDER, we also provide a dataflow query interface that is based on Pig [15]. Pig consists of a high-level language similar to SQL and a compiler, which translates Pig programs to sequences of MapReduce jobs. As such, the jobs resulting from this compilation can run in SLIDER without adjustment. This way, we transparently run Pig-based sliding window computations in an incremental way.

4 Related Work

Next, we present a survey of work in the areas of sliding-window and incremental processing.

Dynamic and data-streaming algorithms. This class of algorithms are designed to efficiently handle changing input data. Several surveys discuss the vast literature on dynamic algorithms, e.g., [5, 18, 22, 38]). Despite their efficiency, dynamic algorithms are difficult to develop and specific to a use case: their adaptation to other applications is either not simple or not feasible.

Programming language-based approaches. Programming languages researchers developed incremental computation techniques to achieve automatic incrementalization [1, 38]. The goal is to incrementalize programs automatically without sacrificing efficiency. Recent advances on self-adjusting computation made significant progress towards this goal by proposing general-purpose techniques that can achieve optimal update times [1]. This work, however, primarily targets sequential computations. The recent work on iThreads [10] supports parallel incremental computation. In contrast, we developed techniques that are specific to sliding-window computations, and operate on big data by adapting the principles of self-adjusting computation for a large-scale parallel and distributed execution environment.

Database systems. There is substantial work from the database community on incrementally updating a database view (i.e., a predetermined query on the database) as the database contents change [17]. database view [17]. In contrast, our focus on the MapReduce model, with a different level of expressiveness, and on sliding-window computations over big data brings a series of different technical challenges.

Distributed systems. There is some work on incremental large-scale processing of unstructured data sets [11–13, 24, 27, 28, 33]. SLIDER is designed to operate at

the same scale and shares some characteristics with this work, such as including a simple programming model, transparent data-parallelization, fault tolerance and scheduling. Furthermore, it builds on some concepts from this prior work, namely the idea of Incoop [13] to break up the work of Reduce task using Combiner functions organized in a tree. There are two important distinctions to these previous proposals. First, prior proposals such as Incoop use memoization instead of change propagation: when the data to a MapReduce computation changes, Incoop scans the whole input data again and performs all the steps of the computation except that it can reuse results from Map and Combine tasks stored via memoization. In contrast, each run in SLIDER knows only about the new data that was introduced to the window and the old data that was dropped, and only has to examine and recompute the subset of the computation that is transitively affected by those changes. Second, our design includes several novel techniques that are specific to sliding window computations are important to improve the performance for this mode of operation. The follow up work on IncApprox [27] extends the approach of Slider to support incremental approximate computing for sliding window analytics. In particular, IncApprox combines incremental computation and approximate computation [34–37].

Batched stream processing. Stream processing engines [16,19,25] model the input data as a stream, with data-analysis queries being triggered upon bulk appends to the stream. These systems are designed for append-only data processing, which is only one of the cases we support. Compared to Comet [25] and Nova [16], SLIDER avoids the need to design a dynamic algorithm, thus preserving the transparency relative to single-pass non-incremental data analysis. Hadoop online [19] is transparent but does not attempt to break up the work of the Reduce task, which is one of the key contributions and sources of performance gains of our work.

5 Conclusion

In this chapter, we present techniques for incrementally updating sliding window computations as the window slides. The idea behind our approach is to structure distributed data-parallel computations in the form of balanced trees that can perform updates in asymptotically sublinear time, thus much more efficiently than recomputing from scratch. We present several algorithms for common instances of sliding window computations, describe the design of a system, SLIDER [8,9], that uses these algorithms, and present an implementation along with an extensive evaluation. Our evaluation shows that 1) SLIDER is effective on a broad range of applications, 2) SLIDER drastically improves performance compared to the re-computing from scratch, and 3) SLIDER significantly outperforms several related systems. These results show that some of the benefits of incremental computation can be realized automatically without requiring programmer-controlled hand incrementalization. The asymptotic efficiency analysis of self-adjusting contraction trees is available [7].

References

1. U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.
2. U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2009.
3. U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Proceedings of the 26th Annual Symposium on Computational Geometry (SoCG)*, 2010.
4. G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.
5. B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Sliding window computations over data streams. Technical report, 2002.
6. P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.
7. P. Bhatotia. Asymptotic analysis of self-adjusting contraction trees. *CoRR*, abs/1604.00794, 2016.
8. P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference (Middleware)*, 2014.
9. P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis. Technical Report MPI-SWS-2012-004, MPI-SWS, 2012. http://www.mpi-sws.org/tr/2012-004.pdf.
10. P. Bhatotia, P. Fonseca, U. A. Acar, B. Brandenburg, and R. Rodrigues. iThreads: A Threading Library for Parallel Incremental Computation. In *proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.
11. P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.
12. P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.
13. P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2011.
14. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2010.
15. C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.
16. C. Olston et al. Nova: Continuous Pig/Hadoop Workflows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011.
17. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1991.
18. Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 1992.
19. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.
20. Costa et al. Camdoop: exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

21. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2004.
22. C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*. Chapman & Hall/CRC, 2004.
23. A. S. Foundation. Apache hive, 2017.
24. P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.
25. B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.
26. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2007.
27. D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *Proceedings of the 25th International Conference on World Wide Web (WWW)*, 2016.
28. D. Logothetis, C. Olston, B. Reed, K. Web, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.
29. D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX ATC)*, 2011.
30. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.
31. D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.
32. D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
33. D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.
34. D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. Privacy preserving stream analytics: The marriage of randomized response and approximate computing. https://arxiv.org/abs/1701.05403, 2017.
35. D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. PrivApprox: Privacy-Preserving Stream Analytics. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2017.
36. D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. Approximate Stream Analytics in Apache Flink and Apache Spark Streaming. *CoRR*, abs/1709.02946, 2017.
37. D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. StreamApprox: Approximate Computing for Stream Analytics. In *Proceedings of the International Middleware Conference (Middleware)*, 2017.
38. G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1993.
39. O. Sümer, U. A. Acar, A. Ihler, and R. Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 2011.
40. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, 2010.

41. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Conductor: Orchestrating the Clouds. In *proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.
42. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
43. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.
44. M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.