Optimistic Recovery for Iterative Dataflows in Action

Sergey Dudoladov[1]
Asterios Katsifodimos[1]

Chen Xu[1]
Stephan Ewen[2]
Volker Markl[1]

Sebastian Schelter[1]
Kostas Tzoumas[2]

[1]Technische Universität Berlin
firstname.lastname@tu-berlin.de
[2]Data Artisans GmbH
firstname@data-artisans.com

ABSTRACT

Over the past years, parallel dataflow systems have been employed for advanced analytics in the field of data mining where many algorithms are iterative.

These systems typically provide fault tolerance by periodically checkpointing the algorithm's state and, in case of failure, restoring a consistent state from a checkpoint.

In prior work, we presented an optimistic recovery mechanism that in certain cases eliminates the need to checkpoint the intermediate state of an iterative algorithm. In case of failure, our mechanism uses a *compensation function* to transit the algorithm to a consistent state, from which the execution can continue and successfully converge. Since this recovery mechanism does not checkpoint any state, it achieves optimal failure-free performance while guaranteeing fault tolerance.

In this paper, we demonstrate our recovery mechanism with the Apache Flink data processing engine. During our demonstration, attendees will be able to run graph algorithms and trigger failures to observe the algorithms recovering with compensation functions instead of checkpoints.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Parallel databases

Keywords

Iterative Algorithms; Fault-Tolerance; Optimistic Recovery

1. INTRODUCTION

In recent years, the growing demand for large-scale data analysis has led to the development of new data-parallel computing platforms like MapReduce [6], SCOPE [5], Apache Flink [1] and Apache Spark [15]. Such platforms on a day-to-day basis execute a variety of data mining tasks ranging from simple grep-style log analysis to complex machine learning algorithms.

[1]Apache Flink originated from the Stratosphere research project [1]. See `https://flink.apache.org`

SIGMOD '15, May 31 - June 04, 2015, Melbourne, VIC, Australia
Copyright 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00
http://dx.doi.org/10.1145/2723372.2735372 .

An important primitive present in many machine learning algorithms is *iteration* (or recursion). Iteration repeats a certain computation until a termination condition is met. The need for efficient execution of iterative algorithms spawned the development of specialized systems [10, 11] as well as integration of iterations into existing dataflow systems [3, 8].

Iterative dataflow computations are often deployed in large clusters of commodity machines, where failures are common. This makes a dataflow system's ability to deal with failures important. The usual approach to fault tolerance is to periodically checkpoint the algorithm state to stable storage. Upon failure, the system restores the state from a checkpoint and continues the algorithm's execution. This method is commonly referred to as *rollback recovery* [7].

This pessimistic approach works well if failures happen regularly. However, real-world use cases indicate that many computations do not run for such a long time or on so many nodes that failures become commonplace [17]. In the case of less frequent failures checkpoints may unnecessarily increase the latency of a computation [16]. Since one still needs to protect against failures, other approaches to recovery may be worth investigating.

In prior work [14], we exploited the convergence properties of certain classes of iterative algorithms to provide an optimistic recovery mechanism. Instead of restoring the lost state from a checkpoint, our mechanism restores the lost state through a *compensation function*. This user-supplied function generates a consistent algorithm state, and the algorithm then resumes execution converging to a correct result as if no failures had occurred.

In this paper, we demonstrate this recovery technique using Apache Flink as a testbed. To exemplify how iterative computations can recover without checkpoints, we employ the Connected Components and PageRank algorithms: Schelter et al. [14] have previously shown them to converge after recovery with compensation functions. We implement a graphical user interface (GUI) with which conference attendees will be able to choose Flink tasks to fail during the algorithms' execution. The failures lead to the partial loss of intermediate results; attendees can observe how compensation functions restore the lost state and how the algorithms converge afterwards.

2. BACKGROUND

Section 2.1 introduces the Apache Flink data processing engine and describes its support for efficient execution of iterative algorithms. Section 2.2 discusses the optimistic recovery of iterative computations with compensation functions.

2.1 Apache Flink

Due to suboptimal performance when executing complex dataflows and iterative computations [2], the MapReduce model has been extended by newly emerging dataflow systems [1, 2, 3, 15, 12]. These systems typically represent a program as a directed acyclic graph (DAG), with vertices representing individual tasks running user-defined functions (UDFs) and edges representing data exchanges among vertices. The systems explicitly support efficient execution of iterative computations in a distributed manner [8, 3, 15, 12].

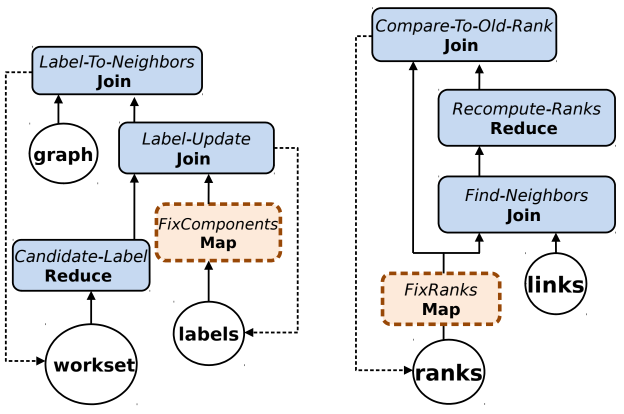
We build upon Apache Flink, which extends the MapReduce paradigm with several higher-order functions such as *Join* (for applying a UDF to the result of an equi-join between two datasets) or *Cross* (for applying a UDF to the cross product of two datasets). A user expresses a data analysis program in terms of these functions and UDFs using the high-level language API. Flink then compiles the program into a DAG of operators, optimizes it and runs in a cluster. Flink allows the user to mark a part of the DAG as iterative. The system then repeatedly executes that part of the DAG by forwarding the output of its last operator to its first operator. The execution finishes when either a predefined number of iterations has been run or a user-supplied termination criterion is met [8].

Flink provides two ways to execute iterative parts of a DAG: bulk iterations and delta iterations. *Bulk iterations* always recompute intermediate result of an iteration as a whole. However, in many cases parts of the intermediate state converge at different speeds, e.g. in single-source shortest path computations in large graphs. In such cases, the system would waste resources by always recomputing the whole intermediate state, including the parts that do not change anymore. To alleviate this issue, Flink offers *delta iterations*. This mode models an iterative computation with two datasets: the *solution set* holds the current intermediate result, while the *working set* holds updates to the solution set. During a delta iteration the system consumes the working set, selectively updates elements of the solution set, and computes the next working set from the updates. The delta iteration terminates once the working set becomes empty.

2.2 Optimistic Recovery

To execute algorithms on massive datasets in a distributed manner, algorithms' intermediate results must be partitioned among machines. Failures cause the loss of a subset of these partitions; to continue execution, the system has to restore the lost data first. Rollback recovery is a popular method to ensure fault tolerance [7]. The idea is to periodically checkpoint the algorithm state to stable storage. Upon failure, the system halts execution, restores a consistent state from a previously written checkpoint and resumes execution. This approach has the drawback that it always incurs overhead to the execution, even in failure-free cases. An alternative is lineage-based recovery [15], which retains lineage, i.e. information about how a partition was derived. After a failure, lineage allows the system to recompute the lost partitions only. Unfortunately, lineage-based recovery does not perform well for many iterative computations, because a partition of the current iteration may depend on all partitions of the previous iteration (e.g. when a reducer is executed during an iteration). In such cases after a failure the iteration has to be restarted from scratch to re-compute lost partitions.

In previous work [14], we proposed to exploit the robust nature of a large class of fixpoint algorithms for an optimistic recovery mechanism. These algorithms can converge to the correct solutions from many intermediate states, not only from the one checkpointed before the failure. We introduce a user-defined *compensation function* which a system uses to re-initialize lost partitions. This function restores a consistent state from which an algorithm can converge. For example, if the algorithm computes a probability distribution, the compensation function has to ensure that probabilities in all partitions sum up to one.



(a) Connected Components. (b) PageRank.

Figure 1: Dataflows with compensations.

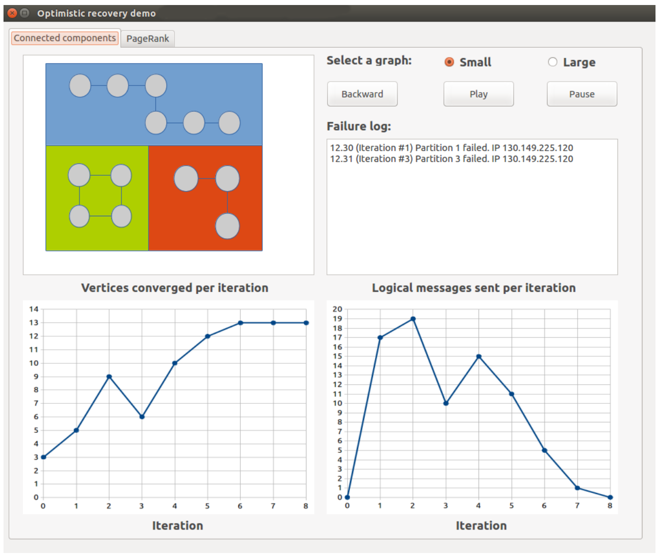
Failure-free execution proceeds as if no fault tolerance is needed: the system neither checkpoints intermediate state nor it tracks the lineage. Therefore, this approach provides optimal failure-free performance. When a failure occurs, the system pauses the current iteration ignoring the failed tasks and re-assigns the lost computations to newly acquired nodes. After that, the system invokes the compensation function on all partitions to restore a consistent state and resumes the execution. Essentially, the compensation function brings the computation "back on track": the function and subsequent algorithm iterations correct errors introduced by the data loss.

To illustrate our approach, we describe how to optimistically recover two well-known fixpoint algorithms via a compensation function.

2.2.1 Recovering Connected Components

The Connected Components algorithm identifies connected components of an undirected graph, i.e. maximum cardinality sets of vertices that can reach each other. We use the diffusion-based algorithm that propagates the minimum label of each component through a graph [9]. Figure 1(a) shows the conceptual Flink dataflow for finding connected components[2] with delta iterations. Initially, we assign a unique label to each vertex (c.f. the 'labels' input which also serves as the solution set). The workset consists of all vertices that updated their labels during the previous iteration; it initially equals to the 'labels' input. The 'graph' dataset contains the edges of a graph. At every iteration,for every vertex we compute the minimum label of its neighbors from the workset via the 'candidate-label' reduce. Next, we compare the resulting candidate labels to the current labels from the solution set in the 'label-update' join. If the candidate label is smaller than the current label, we update the solution set and forward the new label to the 'label-to-neighbors' join. This join computes the workset for the next iteration, which consists of the updated labels and the neigboring vertices of a vertex that was updated. The algorithm converges when there are no more label updates. At convergence,

[2]Blue rectangles denote operators, white circles denote data sources and brown rectangles denote compensation functions. The dotted line around the functions signifies that they are invoked only after failures and are absent from the dataflow otherwise.



Select a graph: [✓ Small] Large

```
12.30 (Iteration #1) Partition 1 failed. IP 130.149.225.120
12.31 (Iteration #3) Partition 3 failed. IP 130.149.225.120
```

Figure 2: GUI for demonstrating optimistic recovery of the Connected Components algorithm.

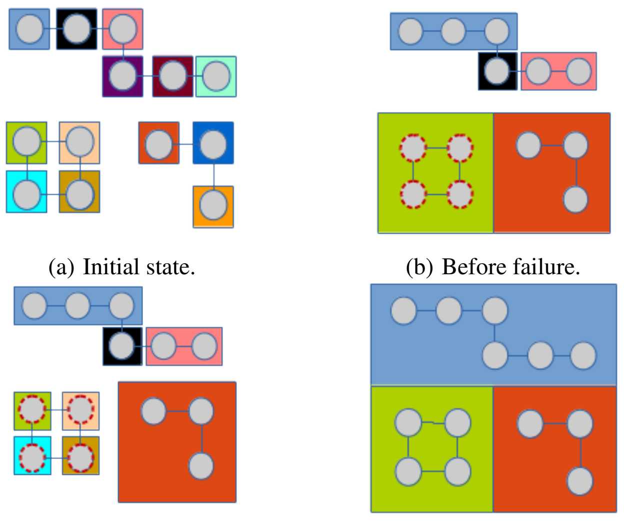


Figure 3: Connected Components algorithm convergence.

all vertices in a connected component share the same label, namely the minimum of the initial labels of vertices in this component.

Compensation function for Connected Components: Failures during the course of the algorithm destroy computed labels for a subset of vertices. Simply re-initializing lost vertices to their initial labels guarantees convergence to the correct solution [14]. The 'fix-components' map executes this compensation function in the dataflow illustrated in Figure 1(a).

2.2.2 *Recovering PageRank*

PageRank is a well-known algorithm for ranking the vertices of a large network by importance with respect to its link structure [13]. The algorithm works by computing the steady-state probabilities of a random walk on the network. Figure 1(b) illustrates the dataflow for modeling PageRank with a Flink bulk iteration. The input consists of the initial vertices' ranks (c.f. the 'ranks' dataset) and the edges with transition probabilities (c.f. the 'links' dataset). During a PageRank iteration, every vertex propagates a fraction of its rank to its neighbors via the 'find-neighbors' join. Next, we re-compute the rank of each vertex from the contributions of its neighbors in the 'recompute-ranks' reduce. At the end of each iteration, we compare the old and new ranks of every vertex to check for convergence (c.f. the 'compare-to-old-rank' join).

Compensation function for PageRank: Losing partitions during the execution of PageRank means that we lose the current ranks of the vertices contained in the failed partitions. As long as all ranks sum up to one, the algorithm will converge to the correct solution [14]. Therefore, it is sufficient to uniformly redistribute the lost probability mass to the vertices in the failed partitions. The 'fix-ranks' map in Figure 1(b) takes care of this.

3. DEMONSTRATION

Section 3.1 introduces the demo infrastructure. Sections 3.2 and 3.3 describe the demonstration scenario and visualization of the failure-recovery process for the Connected Components and PageRank, respectively.

3.1 Setup

The demo setup comprises a laptop and a graphical user interface (GUI). Figures 2 and 4 depict the user interface. By switching the tabs at the top of the interface, users can choose PageRank, if they want to watch the recovery of bulk iterations, or Connected Components, if they want to watch the recovery of delta iterations. Next, attendees pick the input to the chosen algorithm: either a small hand-crafted graph or a larger graph derived from real-world data. Running the demo on the small graph makes it easy to comprehend visually; we slow down the small graph demo so that demo visitors can easily trace each iteration. For the larger graph, we use a publicly available snapshot[3] of the Twitter's social network [4]. We only visualize the small hand-crafted graph in the GUI; for the larger graph, the attendees can track the demo progress only via plots of statistics of the algorithms' execution.

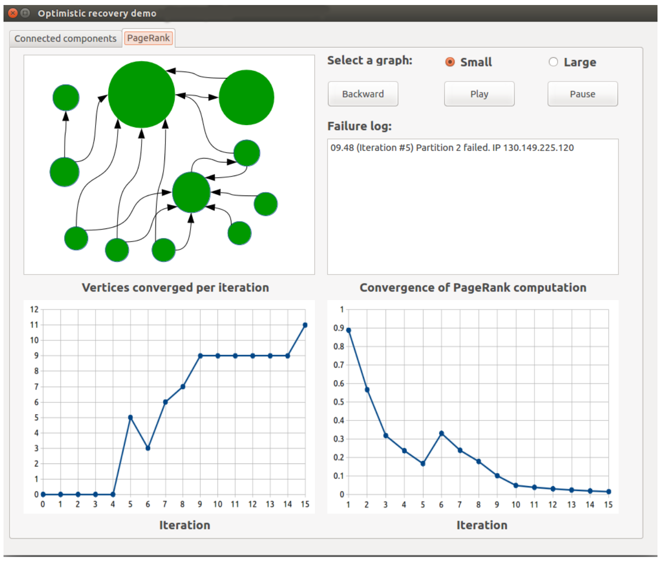
Once the parameters are set, the user presses the "play" button to run the demo. The system then executes the algorithm and visualizes results of each iteration. The "backward" button jumps to the previous iteration; the "pause" button stops the demo at the end of the current iteration. The progression of algorithms on the small graphs is visualized: after an iteration finishes, the interface depicts connected components or page ranks as given by the intermediate results calculated at this iteration.

Conference attendees will be able to choose which partitions to fail and in which iterations via our GUI. The demo tracks such failures and applies the relevant compensation function to restore the lost partitions.

3.2 Connected Components

Figure 2 illustrates the GUI for demonstrating the recovery of Connected Components. Attendees will observe the following demo behavior on the small graph: a distinct color highlights the area enclosing each connected component. Initially, the area around every vertex has a distinct color, as every vertex starts out in its own component (Figure 3(a)). When an iteration finishes, the vertices that changed labels in this iteration are redrawn with a new enclosing color. The new color indicates that the vertices form a new intermediate component. The color comes from the

[3]http://twitter.mpi-sws.org/data-icwsm2010.html



```
09.48 (Iteration #5) Partition 2 failed. IP 130.149.225.120
```

Figure 4: GUI for demonstrating optimistic recovery of the PageRank algorithm.

vertex from which the (updated) vertices accepted their new minimum labels. During the execution, areas of the same color grow as the algorithm discovers larger and larger parts of the connected components . The number of colors decreases; by that attendees can track the convergence of the algorithm. In case of a failure, our GUI highlights the lost vertices (Figure 3(b)), and the compensation function restores them to their initial state (Figure 3(c)). In the end, the algorithm converges and the number of distinct colors equals the number of connected components: the same color encloses all vertices in the same component (Figure 3(d)).

The demo GUI in Figure 2 shows (i) the number of vertices converged to their final connected components [4] at each iteration and (ii) the number of messages (candidate labels send to neighbors) per iteration. The GUI bottom left corner contains the first plot: note the plummet at the third iteration corresponding to the detected failure. Attendees can expect to see similar plummets each time a failure causes a loss of a partition with already converged vertices. The plot at the bottom right corner illustrates the messages sent per iteration. The increased amount of messages at iterations 2 and 4 corresponds to the effort to recover from failures in previous iterations. The system processes more messages compared with a failure-free case, because the vertices restored to their initial labels by the compensation function (as well as their neighbors) have to propagate their labels again.

3.3 PageRank

Figure 4 illustrates the GUI for visualizing the recovery of PageRank. We make the size of a vertex represent the magnitude of its PageRank value: the higher the rank, the larger the vertex. PageRank starts from a uniform rank distribution: all the vertices are of the same size in the beginning (Figure 5(a)). At the end of each iteration, we rescale each vertex proportionally to its recomputed rank. Thus, attendees can watch the convergence of the algorithm: vertices grow and shrink and over time reach their final size, meaning that they converged to their true rank. In the case of a failure, we lose the ranks of the vertices contained in the failing partition. The GUI highlights those vertices (Figure 5(b)), and the compensation function restores their ranks by uniformly distributing the lost probability mass over them (Figure 5(c)). In the end, the vertices converge to their true ranks, irrespective of the compensation (Figure 5(d)).

[4]We precompute the true values for presentation reasons.

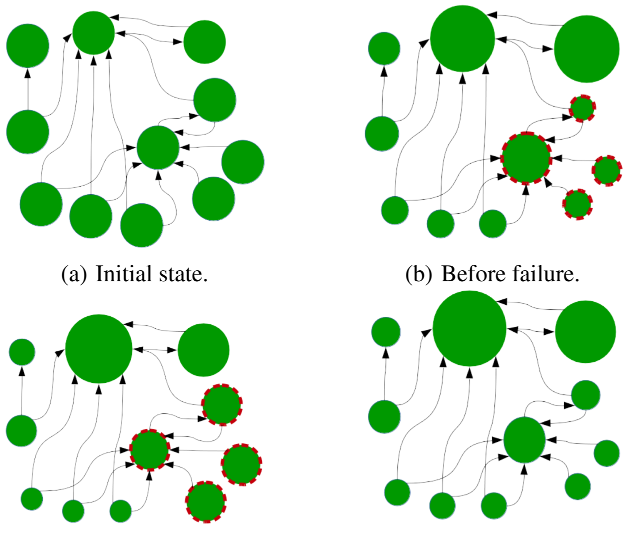


Figure 5: PageRank algorithm convergence.

Analogously to Connected Components, our GUI from Figure 4 plots several statistics collected during the execution: (i) the number of vertices converged to their true PageRank at each iteration and (ii) the convergence behavior of PageRank. The bottom left corner of the GUI shows the first plot. A loss of partitions with converged vertices corresponds to the plummet in the plot in the iteration 6 after the failure in the iteration 5. The second plot (bottom right corner) shows the L1-norm of the difference between the current estimate of the PageRank vector and the estimate from the previous iteration. Over the course of the algorithm, the difference between the estimates at each pair of consecutive iterations becomes smaller and smaller because of the convergence. Hence the downward trend in the plot. Failures appear as spikes in the plot (iteration 6), because the ranks computed during a failure-free iteration are to be closer to the estimates obtained in the previous iteration than the rescaled ranks compensated after failures. Hence, we can expect to observe an increase in the difference after an iteration with failures.

4. ACKNOWLEDGMENTS

The authors would like to acknowledge the invaluable help of Mingliang Qi in developing the GUI of this demonstration. This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere, by the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A) and by the EIT ICT Labs as EUROPA 2014 EITs Cloud-Based Data project (activity code 12115).

5. REFERENCES

[1] A. Alexandrov, R. Bergmann, S. Ewen, J. C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal'14*, pp. 939–964.

[2] S. Babu and H. Herodotou. Massively Parallel Databases and MapReduce Systems. *Foundations and Trends in Databases'12*, 5(1):1–104.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *VLDB'10*, pp. 285–296.

[4] M. Cha, H. Haddadi, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. *ICWSM'10*, pp. 10–17.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB'08*, pp. 1265–1276.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM* '08, 51(1):107–113.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Survey'02*, 34(3):375–408.

[8] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB'12*, 5(11):1268–1279.

[9] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. *ICDM'09*, pp. 229–238.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *VLDB'12*, pp. 716–727.

[11] 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. *SIGMOD'10*, pp. 135–146.

[12] D. G. Murray, F. Mcsherry, R. Isaacs, M. Isard, P. Barham, and S. Valley. Naiad: A Timely Dataflow System. *SOSP '13*, pp. 439–455, 2013.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford Infolab'98*, pp. 1–17.

[14] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. "All Roads Lead to Rome:" Optimistic Recovery for Distributed Iterative Data Processing. *CIKM'13*, pp. 1919–1928.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud'10*, pp. 1–7.

[16] P. Upadhyaya, Y. Kwon, and M. Balazinska. A Latency and Fault-Tolerance Optimizer for Online Parallel Query Plans. *Proceedings of the 2011 International Conference on Management of Data - SIGMOD '11*.

[17] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proceedings of the 2012 VLDB Endowment*, pp. 1802–1813.