Large-scale Incremental Data Processing with Change Propagation

Pramod Bhatotia *Alexander Wieder* *İstemi Ekin Akkuş* *Rodrigo Rodrigues* *Umut A. Acar*

Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Incremental processing of large-scale data is an increasingly important problem, given that many processing jobs run repeatedly with similar inputs, and that the *de facto* standard programming model (MapReduce) was not designed to efficiently process small updates. As a result, new systems specifically targeting this problem (e.g., Google Percolator, or Yahoo! CBP) have been proposed. Unfortunately, these approaches require the adoption of a new programming model, breaking compatibility with existing programs, and increasing the burden on the programmer, who now is required to devise an incremental update mechanism. We claim that automatic incremental processing of large-scale data is possible by leveraging previous results from the algorithms and programming languages communities. As an example, we describe how MapReduce can be improved to efficiently handle small input changes by automatically incrementalizing existing MapReduce computations, without breaking backward compatibility or demanding programmers to adopt a new programming approach.

1 Introduction

Large-scale processing of unstructured data sets is an increasingly common and important task for Internet services, from e-commerce to social networking sites. In this context, MapReduce [5] has emerged over the last few years as the *de facto* standard programming model for this kind of processing. This model and the associated run-time system was originally adopted by Google, and, subsequently, an open-source platform named Hadoop, which supports the same programming model, has also gained tremendous popularity.

Despite this success, Google has recently proposed an alternative system called Percolator, which is aimed at processing incremental updates to large data sets [11]. The motivation behind proposing a different system is that many data processing tasks operate on an input set that changes incrementally, and MapReduce was not designed to process small incremental updates. Rather, MapReduce was optimized to process large batches of data efficiently [11]. Percolator uses an alternative programming model where the user is asked to implement an efficient *incremental-update mechanism* by defining upcalls. These upcalls are triggered by changes in user-defined portions of the data, and update the output accordingly. Since the work done is often proportional to the update size, rather than the total input size, Percolator achieves huge performance gains. A system called CBP, with a similar philosophy, was proposed by Yahoo! [9].

A downside of these new proposals is that, by departing from the current programming models for large-scale data processing, these systems not only lose compatibility with the existing code base of MapReduce programs, but also shift the burden of designing and implementing an efficient incremental-update mechanism to the programmer. The problem associated with this shift goes beyond the need to adopt a new programming model: it lies also in the fact that developing efficient algorithms for incremental computations is considerably more difficult than writing algorithms for a static input data set. To give a simple example, consider how you would compute the minimum element in a list where elements can be added or removed. For a static list of input data elements, the minimum element can easily be computed in linear time ($O(n)$, where n is the number of elements in the list). However, recomputing the minimum element from scratch for every insertion or deletion is a costly operation, especially for large n. To circumvent this linear time complexity, additional data structures like a *minimum heap* could be maintained to provide a logarithmic bound ($O(log\ n)$). Furthermore, when we move beyond such simple examples, the development of algorithms for incremental computations is a complex, application-specific task that has fueled an entire research area [4, 6].

In this paper, we take the position that automatic and efficient incremental data processing at large scale could be achieved by combining ideas and techniques developed in algorithms, programming languages, and

distributed systems research. The resulting techniques would neither require a radical departure from current models of programming, nor to invent and implement complex algorithms for efficient incremental computations, nor to program distributed systems.

As evidence for the feasibility of our position, we propose an approach for transparently and automatically incrementalizing existing MapReduce programs without requiring the user to change any code. The idea behind this approach is to shift the burden of reasoning about how to efficiently process incremental updates from the programmer to the system itself, combining advances from the algorithms, languages, and systems communities to perform efficient and transparent updates. Our proposal is to extend the MapReduce framework with *change propagation*, where the framework keeps track of the dependencies between subsets of each MapReduce computation, and, upon changing of a subset of the input, rebuilds only the parts of the computation and the output affected by the changes. We present a high-level sketch of an extension to the Hadoop architecture to support change propagation, highlighting the main technical challenges and sketching ideas for overcoming them.

Finally, we perform a proof-of-concept evaluation of the potential benefits of our proposal. This evaluation resorts to a single-node implementation of MapReduce, to which we add the ability to perform change propagation. Our evaluation indicates that the performance benefits of employing change propagation in MapReduce programs can be huge. This not only demonstrates the possibility to add the capability of incremental processing to MapReduce programs in a way that is efficient and transparent, but also our experience with MapReduce can be seen as a first step towards a broader goal of bringing automatic incrementalization to large-scale data processing systems and applications.

2 Incremental Computation Approaches

We review previous work on incremental computations and discuss its relevance to incremental large-scale data processing. For comparing previous approaches, we use three metrics: *expressiveness*, *efficiency*, and *programmability*. Expressiveness refers to the ability to express different kinds of computations (e.g., data-parallel, purely functional, imperative). Efficiency refers to both asymptotic and practical efficiency. Programmability refers to the ease of programming, which we measure by considering the aspects of algorithmic complexity (i.e., how easy/difficult it is to design an efficient incremental algorithm), and how complex the resulting implementations are expected to be. Table 1 shows a summary evaluation of some approaches with respect to these metrics.

Algorithms community. In this community, researchers designed so-called *dynamic algorithms* that permit dynamic changes to their input, and update their output upon these modifications. Several surveys illustrate the vast literature on dynamic algorithms [4, 6]. This research shows that dynamic algorithms can be asymptotically, often by a near-linear factor, more efficient than their conventional counterparts. In large-scale systems, this asymptotic difference can yield huge speedups. Although dynamic algorithms are expressive (an algorithm being the basic way in which a solution to a problem can be expressed) and highly efficient, they can be difficult to develop and implement even for simple problems; some problems took years of research to solve and many remain open. Furthermore, there is no methodology one can prescribe that suits all problems: a new algorithm must be devised for each computation.

Programming languages community. In this community, researchers developed so-called *incremental computation* techniques to achieve automatic incrementalization. The basic idea is to automatically translate a program that is conventionally defined as mapping between an input and an output to a program that allows modifications to its input while updating its output. Incremental computation attracted significant attention and many techniques have been proposed (e.g., [13]). Being automatic and hiding the mechanism for incrementalization, this approach simplifies software development. Self-adjusting computations extended these techniques to allow for expressing incremental computations at a high-level and deriving efficient executables by using compilers specifically developed for this purpose [8]. State-of-the-art incremental computation techniques are expressive (applicable to both functional and imperative computation models), efficient (optimal updates are often possible), and can be easy to use. Nonetheless, in certain cases, the approach can require algorithmic reasoning to achieve optimal complexity bounds, which can increase algorithmic and software complexity, and require program modifications.

Systems community. Practitioners have built several systems based on memoization (e.g., Haloop [2], Nectar [7], DryadInc [12]). These approaches are a first step towards obtaining a solution that does not require the programmer to devise and implement a dynamic algorithm. However, their efficiency can still be improved, since they do not maintain a dependency graph (as we explain next). Contrary to the memoization-based approach, some proposed systems require the programmer to devise a dynamic algorithm [9, 11]. In particular, Google's Percolator [11] adopts an event-driven programming model, where an application is structured as a series of observers. Observers are triggered by the system whenever user-specified data changes. Observers in turn can modify other data forming a dependence chain that implements the incremental data processing. Sim-

Approach	Expressiveness	Efficiency	Programmability	
			Algorithmic Complexity	Implementation Complexity
Algorithms	High ✓	High ✓	High ✗	High ✗
Programming Languages	High ✓	High ✓	Low—High ✓✗	Low—High ✓✗
Percolator, CBP	High ✓	High ✓	High ✗	High ✗
Incremental Data Proc.	**High ✓**	**High ✓**	**Low ✓**	**Low ✓**

Table 1: Approaches to incremental problems in algorithms and programming languages community, recently proposed systems for large-scale data processing. Incremental data processing should be able to combine the best features of these approaches.

ilarly, Yahoo!'s continuous bulk processing (CBP) [9] proposes a data-parallel programming model along the lines of MapReduce by introducing new primitives to store and reuse prior state for incremental processing, where *loopback flows* are used to redirect the output of a stage as its input. Both systems are expressive and can be efficient, as they give the programmer full control over how modifications to the input are handled. However, they have high algorithmic and implementation complexity, since the programmer must implement the efficient incremental update algorithm.

Database community. Several techniques were proposed for performing incremental view maintenance, i.e., efficiently updating the results of a pre-defined database query upon changes to table contents [3]. These techniques have been engineered to be highly efficient, and impose no extra burden on the programmer, since the database engine hides all the complexity of the incrementally updating query results. However, in terms of their expressiveness, these techniques are geared towards database systems (or, more recently, large-scale relational query processing framework [10]) and therefore their interface is limited to the query language supported by the database engine.

3 Automatic Incrementalization

Prior advances on automatic incrementalization (Section 2) suggest that large-scale incremental data processing should be possible. If successful, this approach will simplify both software and its development, while improving efficiency. Yet, there are several challenges that must be overcome to achieve automatic incrementalization of large-scale data processing applications. We outline these challenges and point out why they seem tractable and how we might address them.

Expressiveness. Existing automatic incrementalization techniques allow expressing arbitrary sequential computations, with some preliminary work on parallel systems. An important challenge will be the generalization of these techniques to allow for expressing parallel and distributed computations. In the cases where explicit use of parallelism or concurrency is not needed, e.g., in MapReduce, the task will be easier because only the techniques under the hood need to be aware of the parallelism and concurrency.

Efficiency. Existing approaches to incremental computation often target shared memory, uniprocessor machines, and tend to make liberal use of memory to save time [8]. In the context of a distributed computation, where the memory-time tradeoff is different, these approaches could lead to disastrous efficiency, calling for careful engineering of the systems, particularly to minimize inter-node data movement.

Programmability. Despite the benefits of automatic incrementalization, a generic updater can sometimes result in suboptimal outcomes, making custom solutions necessary. Recent results show that these two approaches can be combined modularly [1]. We thus should be able to create a library of custom solutions to be employed by the programmer on a needs basis. Depending on the specifics of data processing domains, we may develop domain-specific languages and systems to take advantage of that domain's structure.

4 Change-propagating MapReduce

In this section, we support the case for a new approach to large-scale data processing by sketching how we can adapt the change propagation mechanism of self-adjusting computations [8] to MapReduce. The resulting framework preserves the expressiveness of conventional MapReduce, and can respond to small incremental changes to data asymptotically more efficiently than a complete re-computation. This is done automatically, with no additional programming effort. We therefore achieve expressiveness, efficiency, and programmability.

Basic approach. The principle behind self-adjusting computations is to track all computation data and all sub-computations that depend on them in a *dynamic dependency graph* and use a change propagation algorithm, which, given an update, identifies the sub-computations that depend on the data and re-executes them. This may modify other data, causing other sub-computations to be re-executed. Change propagation terminates when all modified data and their dependent sub-computations complete. In language-based approaches to self-adjusting computation [8], the construction of the

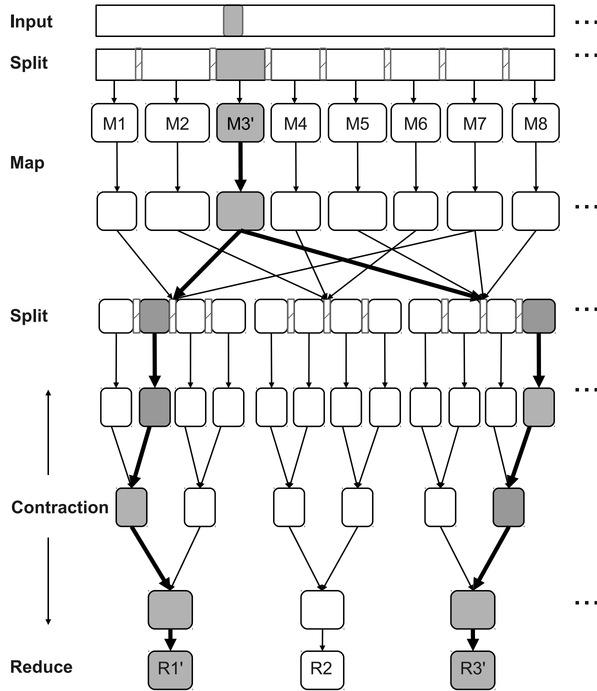

Figure 1: An example computation graph of MapReduce and change propagation. An input change affects only the shaded tasks via dependency edges, and the remaining ones are reused.

computation and the change propagation are supported automatically by using a combination of compilation and algorithmic techniques. Here, we specialize these techniques to the MapReduce framework by taking advantage of our knowledge of the computation structure.

Contraction phase and stable MapReduce. For change propagation to be efficient it is critical to strike a balance between 1) granularity: the computation should be split into tasks that are small enough, and 2) shallowness of dependence graph: the chains of dependency between computations should be short. Additionally, these computations should be *stable*: the structure of their computation graph should not change dramatically as their input data is modified by a small amount. Since the MapReduce framework is naturally parallel, it has short dependency chains. It can, however, yield unstable computations due to the first requirement. The problem is that, when we consider the smallest unit of computation to be Map and Reduce tasks, we observe that we can arbitrarily control the granularity of Map tasks by varying the size of the input chunk that is assigned to each Mapper, but, for Reduce tasks, this cannot be done, since their granularity depends only on the number of values that were associated with each key that was emitted as an output by the Map phase. In other words, a Reducer with a large number of values will be too coarse-grained.

To address this, we leverage an already existing mechanism for controlling the granularity of processing tasks, but with a different purpose. In particular, we propose taking advantage of Combiners, which were originally proposed to reduce network traffic by anticipating a small part of the processing done by Reduce tasks. Given this, and inspired by an algorithmic idea, list-contraction, we add another phase called the *Contraction* phase between Map and Reduce. In this phase, after collecting the records belonging to the same key, we split each input to a Reduce task into smaller chunks and contract the tasks by constructing a balanced tree on them. Each level of the tree applies the combiner function to a subset of the nodes, until we reach the root of the tree, where the outcome of the Contraction phase is fed to the Reducer as in conventional MapReduce.

Example. Figure 1 shows an example computation graph for a MapReduce job with change propagation. Note that the contraction phase allows for splitting the results previously generated by Reduce tasks, yielding a shallow computation graph with logarithmic depth, which allows for efficient change propagation. Also, going back to the example in the introduction about computing the minimum, the contraction phase is what allows the incremental computation to be transparent and only incur logarithmic cost.

Change propagation. Given modifications to data, the change propagation mechanism first updates the structure of the computation graph, introducing new tasks and deleting existing tasks as necessary. Change propagation then determines the tasks that are reachable from the changed data and re-executes them. Since the computation graph is shallow, such a propagation requires a very small number of tasks to be re-executed. For example, if only one record is changed, no more than a logarithmic number of tasks will need to be re-executed.

Task granularity. In order to determine the right task granularity a tradeoff between efficiency and space overhead has to be made: a smaller task size will make the change propagation more efficient, but on the other hand also increases the space overhead for storing the intermediate results. In practice, choosing a task size that is too small will also result in high scheduling and communication overheads.

Input mechanisms. One of the challenges in extending the Hadoop framework is that we need a data storage that permits changes to data stored in files. We envision three options to address this. First, we can restrict changes to adding new content to the input by appending to an HDFS file. Second, we can consider that the input for consecutive runs is stored in separate HDFS files. This would require detecting the differences between the two files efficiently. The third alternative is to store the input to each task in HBase instead of HDFS. HBase supports mutations to elements stored in each table and is also supported by Hadoop.

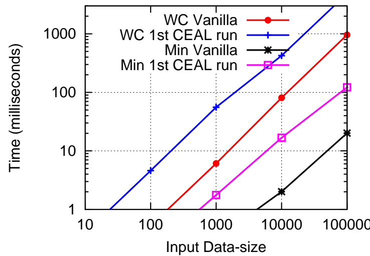

Figure 2: Initial run times (without change propagation and incremental).

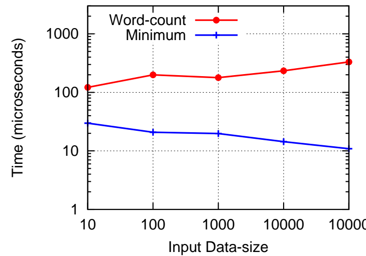

Figure 3: Change-propagation times.

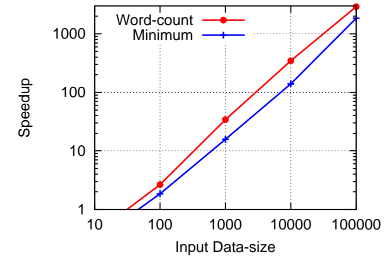

Figure 4: Speedup of change propagation.

5 Evaluation

We implemented a proof-of-concept prototype to test the feasibility of incorporating change propagation to a MapReduce framework. Our prototype leverages CEAL [8], a C complier targeting sequential, shared-memory machines, which generates executables that automatically respond to modifications to their data using change propagation. Our prototype is a single-node implementation of MapReduce in CEAL, enabling automatic incremental updates in MapReduce applications. We developed two applications for our framework: a word-count application and an application that finds the minimum in a sequence of numbers.

In our experiments, we randomly generate the input and vary the input size in each run. For the incremental version of MapReduce, we measure two characteristics: the runtime for initially processing the full input, and the runtime for propagating a single input change to the final result. For comparison, we use the same implementation with the change propagation functionality disabled.

In Figure 2, we show the absolute runtimes for both applications, with change propagation disabled (`Vanilla`) and CEAL's first time run when varying the input data size. In the case of CEAL's first run, the full data has to be processed because no computation results from previous runs can be leveraged. As a result, setting up the initial data structures for change propagation imposes runtime overhead only for the first time. Figure 3 depicts the absolute runtime for processing incremental changes when change propagation is enabled. When comparing the runtime graphs, we see that the change propagation mechanisms break the linear dependency of runtime on input size for incremental processing tasks. In Figure 4 we show the asymptotic speedup gains of change propagation over reprocessing the full input.

6 Conclusions

In this paper we take the position that the MapReduce model can be extended to efficiently automatically process incremental updates, by leveraging approaches from other communities. Our experimental evaluation of a proof-of-concept, single node prototype has revealed a large potential for performance improvements.

We see our proposal as contributing to help bridge the gap between easy-to-use solutions for bulk data processing that do not support incremental computations, and approaches for change propagation developed in the algorithms and programming languages communities. Our aim is to take a step further in the direction of a symbiotic approach that inherits the best of both worlds.

References

[1] ACAR, U. A., BLELLOCH, G. E., LEY-WILD, R., TANGWONGSAN, K., AND TÜRKOĞLU, D. Traceable data types for self-adjusting computation. In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)* (2010).

[2] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. HaLoop: Efficient iterative data processing on large clusters. In *36th International Conf. on Very Large Data Bases* (Sep. 2010).

[3] CERI, S., AND WIDOM, J. Deriving production rules for incremental view maintenance. In *Proc. 17th International Conference on Very Large Data Bases* (1991).

[4] CHIANG, Y.-J., AND TAMASSIA, R. Dynamic algorithms in computational geometry. *Proceedings of the IEEE 80*, 9 (1992).

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*.

[6] DEMETRESCU, C., FINOCCHI, I., AND ITALIANO, G. *Handbook on Data Structures and Applications*. 2005, ch. 36.

[7] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in data centers. In *Proc. OSDI 2010*.

[8] HAMMER, M. A., ACAR, U. A., AND CHEN, Y. CEAL: a C-based language for self-adjusting computation. In *Proc. Conf. Programming Language Design and Implementation (PLDI'09)*.

[9] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *Proc. 1st Symp. on Cloud computing (SoCC'10)*.

[10] OLSTON ET. AL., C. Nova: Continuous pig/hadoop workflows. In *Proc. Intl. Conf. on Mgt. of Data (SIGMOD 2011)*.

[11] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *9th Symp. Operating Systems Design and Implementation (OSDI'10)*.

[12] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *Worksh. on Hot Topics in Cloud Computing (HotCloud'09)*.

[13] RAMALINGAM, G., AND REPS, T. A categorized bibliography on incremental computation. In *Principles of Programming Languages* (1993), pp. 502–510.