DryadInc: Reusing work in large-scale computations

Lucian Popa *Mihai Budiu, Yuan Yu, Michael Isard*
UC Berkeley *Microsoft Research, Silicon Valley*

Abstract

Many large-scale (cloud) computations operate on append-only, partitioned datasets. We present two incremental computation frameworks to reuse prior work in these circumstances: (1) reusing identical computations already performed on data partitions, and (2) computing just on the newly appended data and merging the new and previous results.

1 Introduction

One of the most successful applications of cloud computing is the analysis very large data sets. Batch processing platforms such as Google's Map-Reduce [3] and Sawzall [16], Yahoo's Hadoop [18] and Pig Latin [15], and Microsoft's Dryad [11], DryadLINQ [19] and Scope [2] have been developed for this purpose. Many, if not most, of the computation cycles expended by these engines are currently employed for analyzing logs, such as search engine logs. The analysis of very large scientific data sets (e-Science) [7] is another emerging class of applications suitable for these platforms.

An interesting common feature of these applications is that the input data (a) continuously grows and (b) old data does not change[1]. Even the storage systems developed for storing such data (The Google File-system [6], HDFS [18] and Cosmos [2]) take advantage of the append-only nature of these large data sets. Although the logs change only incrementally, many useful computations need to process repeatedly the entire data set.

In this paper we are investigating the problem of *incrementalizing* the computation as well: given a computation of a large data set, we attempt to perform it efficiently on an incrementally larger data-set, reusing most of the effort. This leads to faster executions, higher cluster throughput and less energy consumed. We are interested in finding solutions which automate this task as much as possible.

The space of incremental computations presents several non-trivial trade-offs: (1) particular versus generic – a custom solution can be much more efficient than a generic one by taking advantage of the problem semantics; (2) automatic versus manual; (3) time versus space –incremental computations require the storage of previous and/or partial results which are reused; and (4) safety versus efficiency – there are many practical obstacles to defining precisely the "sameness" of two computations, and a very strict notion of "sameness" may prevent reuse.

While we are not claiming to provide a definitive solution, we are exploring in this paper two interesting points of the design space.

Our first solution is called *Identical Computation* (IDE), and is fully automatic. IDE is a form of memoization, which caches partial results and reuses them if they reoccur unchanged in the context of future computations. The second solution is called *Mergeable Computation* (MER), and it requires some support from the user: the programmer has to provide a *merging* function which combines the results computed on an old version of the input with the results computed on the additional input data (delta). Intuitively, IDE is similar to the Unix *make* tool, which avoids recomputing partial results that have not changed, while MER is similar to the Unix *patch* tool, which "fixes" the output given incremental changes in the input.

We have implemented our solution in the context of the Dryad [11] large-scale computing system (described in Section 2). Our solution depends on properties of Dryad computations, which hold for other mainstream computational models as well (such as Map-Reduce); most notably, we rely on the fact that computation is composed from a collection of purely functional processes – in consequence, each process is idempotent and deterministic, and it produces the same outputs when re-executed with the same inputs.

The contributions of this paper are: (1) we present two incremental computation algorithms for the context of large-scale distributed systems; (2) we discuss practical issues for implementing these algorithms in real systems; and (3) we provide a preliminary evaluation of our algorithms in a real implementation.

2 Background and Model

Dryad [11] is a computational model which allows programmers to express distributed batch computations as collections of processes connected via point-to-point channels. The computation is a graph: the graph vertices are processes, and the graph edges are the communication channels. Dryad constrains the computation graph to

[1] We will ignore the privacy requirements which may cause the heads of the logs to be discarded after a bounded retention period.

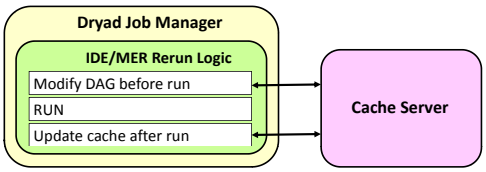

Figure 1: Incremental Computation System Architecture

be acyclic (a DAG). Dryad computations can be seen as sets of *stages*; all the vertices in a stage are performing the same computation. The output of any stage is a collection of records, partitioned into disjoint pieces. We represent the global input and output of a Dryad job with a special kind of stage, composed just from "storage" vertices. We denote inputs by I and outputs by O; input partitions are I1, I2, (e.g., Figure 2).

We assume that the job input and output data is stored on a persistent storage system, similar to the Google File System [6] or Cosmos [2]. Each file is stored as a sequence of disjoint extents, representing disjoint parts of the data. We also call these extents *partitions*. Furthermore, we assume that stored data is *immutable*, and can only change by the addition of new complete extents. We denote the "new" extents by delta, or Δ. The content of each extent is protected with an MD5 checksum; we call these checksums *fingerprints*. The storage system maintains such fingerprints to detect data corruption using scrubbing.

Our algorithms have been tested with Dryad computations jobs that change dynamically during job execution (Dryad supports runtime job graph changes in restricted ways).

3 System Architecture

Figure 1 presents the architecture of our system, which is formed by two components: (1) the *rerun logic*, an extension of the Dryad Job Manager[2], which detects reused computation and performs job graph rewriting and (2) the *cache server*, a network service with a put/get interface.

The Rerun Logic: In the Dryad system job graphs are generated programmatically (i.e., the user uses an API to construct job graphs with arbitrary acyclic shapes). The rerun logic intercepts the job DAG after it has been generated, just prior to its execution. The rerun logic performs the following actions:

(1) Analysis: based on the job DAG and the method used (IDE or MER), this step identifies a set of previous results that may be present in the cache. These results correspond to channels from previously executed DAGs (only channels implemented as files are considered[3]). The rerun logic then checks the cache server for the presence of these results.

(2) DAG rewriting: If some of the results identified at step (1) are found in the cache, the rerun logic modifies the job DAG according to the IDE or MER algorithms, to reuse these results.

(3) Running: Dryad executes the DAG as a regular job.

(4) Caching: After the successful completion of a job, the rerun logic selects partial results that may be useful for future computations and inserts them into the cache. Services of the distributed storage system are called to persist the contents of temporary channels.

Throughout this paper we evaluate IDE and MER in isolation; see Section 7 about combining them.

As a safety measure, the rerun logic always keeps the unmodified Dryad job graph as a back-up for re-execution, in case some of the cached data turns out to be unavailable.

The Cache Server: is a generic cluster-level service with a put/get API operating on key-value pairs. The typical use of the cache server is mapping fingerprints of computations to persistent storage extents. The cache server can implement arbitrary cache replacement policies; the distributed file system garbage-collects the unreferenced extents.

4 Identical Computation (IDE)

IDE stores and reuses the results of computations already performed in the past. We call these *identical computations*. In our context, we reuse computations at the vertex granularity; two vertices are identical if they execute the same code on the same input data. This notion naturally extends to entire computational (sub)DAGs.

To identify identical computations, we extend *fingerprints* to handle computations, not just data. For example, the fingerprint of a vertex computation captures information about the executable invoked, input channels, and the environment of the process, including start-up arguments. Engineering the fingerprints involves some trade-offs between conservativeness and efficiency; for example, some changes in the environment (e.g., the current time) or in the executable (e.g., a program re-linked after a bugfix) may not change the computation behavior.

We further extend fingerprints to also capture the structure (computation and data) of a Dryad DAG. We associate a fingerprint with any vertex and channel in a Dryad job. The fingerprint is computed recursively on the job DAG structure: the fingerprint of a vertex is a function of the fingerprints of its input channels and the fingerprint of the binary executed; the fingerprint of a channel is a function of the vertex which writes to the channel, and of the output channel number (a vertex can have many output channels). Intuitively, the fingerprint of a channel

[2]The Dryad job manager is a centralized process which generates the computation DAG and oversees its execution.

[3]Dryad supports other types of (non-persistent) channels, such as TCP pipes and in-memory FIFOs.

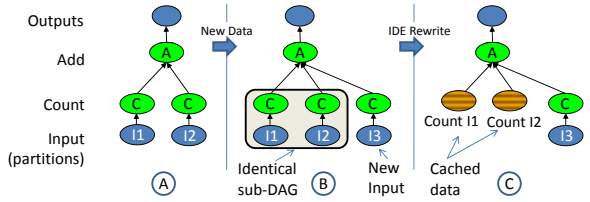

Figure 2: IDE applied to a record-counting application.

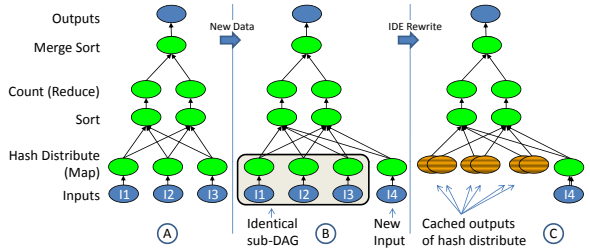

Figure 3: IDE applied to a histogram application.

summarizes the effects of the complete sub-DAG that can influence the channel data. Two channels with the same fingerprint are guaranteed to contain identical data.

Note that IDE can reuse computations between unrelated jobs (and not just between multiple instances of the same job executed on incrementally larger data).

Figure 2 shows a simple example of using IDE on an application that counts the records in a partitioned input file. The first execution (Figure 2A) is performed on an input file with two data partitions. Each `C` (`Count`) vertex performs the count on one partition; the `A` (`Add`) vertex aggregates the individual partition counts. Subsequently, the input file grows to three partitions (Figure 2B). The `Count` vertices operating on the I1 and I2 partitions produce the same results as in the previous execution. If their outputs have been cached, IDE can reuse them by rewriting the computation as shown in Figure 2C. The first two input channels of the `Add` vertex are replaced with stored data containing the cached outputs of the `Count` vertices from the Figure 2A execution.

In general, given two computational DAGs, $G_1 = F(I)$ and $G_2 = F(I + \Delta)$, where $+$ denotes concatenation, IDE locates a common sub-DAG C of both graphs and replaces the instance of C in G_2 with the outputs of C computed and cached in G_1. This method provides most savings when C is the largest common subgraph of G_1 and G_2. Figure 3 shows the original and IDE DAG of an application that computes the histogram of records in a partitioned file.

Implementation: IDE is fully automatic and transparent to Dryad users; the users submit unmodified Dryad jobs, and the system rewrites them to compute only incremental results if the cached data is available.

A challenge for IDE is the choice of the channels to cache after a successful execution; the system has to make this choice before knowing the shape of future computations. Caching all channels is impractical, so we have implemented a heuristic to select a small set of channels to cache. Intuitively, the heuristic attempts to discover a stage where all vertices are affected by Δ; the inputs to this stage are cached. The heuristic marks vertices using a breadth-first traversal of the DAG starting from one random input partition. This process stops when marked vertices isolate inputs from outputs (one cannot create a path of unmarked vertices from an input vertex to an output one). The input channels of the vertices on the frontier of the marked region are cached. For the example in Figure 3 the heuristic chooses to cache the outputs of the hash-distribution stage. In all applications that we have investigated this heuristic has provided optimal results (discovering the maximal common sub-DAG).

For IDE, the analysis phase of the rerun logic applies the heuristic to determine a cut in the DAG and computes the fingerprints for the channels in this cut. The DAG-rewriting phase replaces each channel found in the cache with its associated data partition from the cache, and then removes the resulting dead code. Finally, the caching phase inserts into the cache the channels selected by the heuristic and not already cached.

5 Mergeable Computation (MER)

We define a function $F : I \rightarrow O$ to be *mergeable* if there exists a function $M : O \times O \rightarrow O$, s.t. $F(I + \Delta) = M(F(I), F(\Delta))$[4]. In this paper we consider F to represent the entire computation (the result produced by the output stage), but our approach can be extended to work with intermediate results.

MER caches $F(I)$; given $I + \Delta$, MER only computes $F(\Delta)$, and uses the merge function to compute $F(I+\Delta)$. The programmer has to write the merge function M. MER automatically identifies Δ, detects whether usable previous results exist and synthesizes the incremental DAG $F(\Delta)$. MER can be potentially more efficient than IDE in reusing computation, because it can reuse much more than just common identical sub-DAGs.

Figure 4 shows the effect of applying MER to the record counting application presented earlier. Fig. 4(A) and (B) show the computation on a 2-partition and 3-partition input respectively. Fig. 4(C) shows the result provided by an ideal implementation of MER: the DAG reuses the previous result and adds the count computed on the delta. In this case the merge function is just integer addition; the `Add` vertex itself implements the merge function.

Implementation: To identify whether a previous instance of the application is present in the cache, MER has to verify the following conditions: (1) the executable code that generates the job DAG has not changed from the previous execution, (2) the job executes on the same input file

[4]A more general definition for mergeable functions, which we do not explore here, is $F(I + \Delta) = M(F(I), \Delta)$.

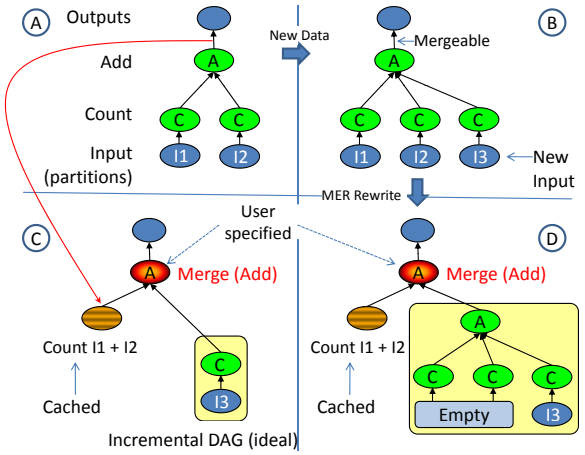

Figure 4: MER applied to a record-counting application.

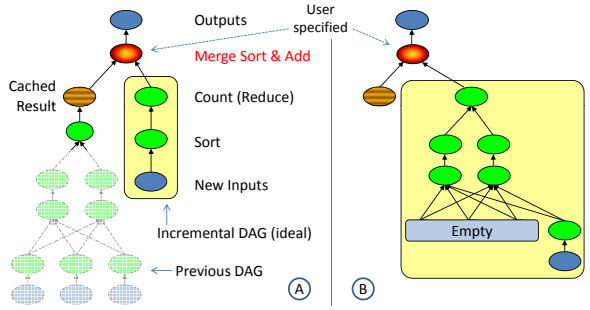

Figure 5: MER applied to a histogram application.

and (3) the code of all vertices involved in the computation has not changed. MER ensures these conditions by computing fingerprints. For simplicity, in the current implementation, we ensure conditions (2) and (3) by restricting that $F(I)$ is isomorphic to a subgraph of $F(I + \Delta)$ (this is safe, but not optimal). MER stores the serialized job DAGs into the cache server; the check for subgraph isomorphism with unchanged vertices and inputs is performed using the same fingerprints as used by IDE.

To identify Δ, MER uses the serialized DAG of the previous execution from the cache.

To synthesize the incremental DAG for $F(\Delta)$, MER substitutes all partitions of I with empty partitions in $F(I + \Delta)$, i.e. $F(\Delta) = F(I + \Delta)|_{I=\phi}$. Applying this algorithm to the record-counting application results in the DAG in Figure 4(D). The resulting $F(\Delta)$ can be improved by using semantic information, e.g., removing vertices that produce empty outputs when given empty inputs, by replacing them with empty partitions, etc.

Fig. 5 shows the result of applying the MER algorithm to the histogram computation example, presented earlier in Fig.3. Fig. 5(A) shows the optimal way to perform the incremental computation when Δ is a single partition. Fig. 5(B) depicts the implementation actually built by our algorithm, (which removed the hash-partitioning vertices with empty inputs). Note that even if vertices with empty inputs are not removed from the graph, the time to execute

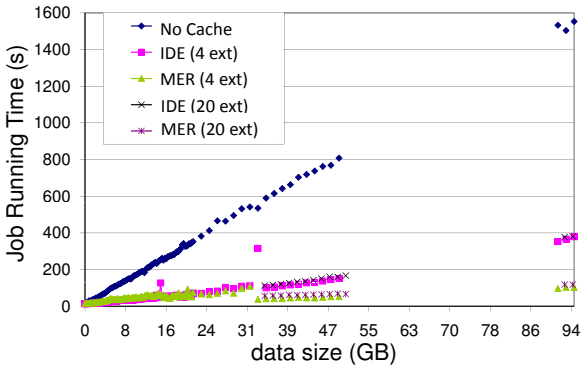

Figure 6: Absolute Running Time (Histogram app.)

them is very likely small.

6 Evaluation

Incremental computation can translate into *faster execution*, *higher cluster job throughput* and *reduced energy consumption*. On the downside, incremental computations require more storage, for caching intermediate results. The correct trade-off depends on many factors; for applications which perform large data reductions the storage trade-off may be very effective. Note that the required cache storage size does not depend on the number of executions of a job and is, in general, less than the size of the input data. The time to compute fingerprints is negligible compared to the job execution and scheduling.

We evaluate IDE and MER on a 8 node cluster, using a single application that computes a histogram of file records (we use the query histogram application from Section 6.3 in [11], a job similar, but more complex than the one in Figure 3). The machines are running 64-bit Windows Server 2003, are equipped with dual-core processors and 8 GB of memory, and have four 400GB disks each, used in a RAID 0 configuration.

Fig. 6 presents the execution times for the histogram application when increasing the input file by four extents at a time (250MB) or 20 extents at at time (1.3GB). The measurements were carried in increasing order of input size, using the cached data from the previous run to start the next run. We skipped input sizes in the range 50-90GB to speed up evaluation (but all shown data points use the same amount of incremental data).

In this example, both MER and IDE provide significant savings (up to 80-90%) and the execution time is essentially linear in the input size. For this application, MER outperforms IDE for large data. There are two reasons: (1) more computation is saved by MER and (2) MER takes advantage of a large reduction size for reused results. In fact, MER runs almost in constant time for very large inputs (the duration is a function of $|\Delta| + |O|$ (output), and after a while $|O|$ stops increasing). However, note that changing some parameters of the job graph (e.g., number

of partitions per stage) can influence the performance: we saw variations which improved IDE by 50% while slowing down MER by 30% (results not shown). The small dip above 30 GB of input for both IDE and MER is caused by us turning off the interactive visualization of the job DAG; the DAGs at this size are large enough to consume most of the CPU of the job manager machine for computing the graph layout. The two sudden spikes in performance for IDE likely represent transient effects in the cluster.

7 Discussion and Related Work

Map-Reduce: Our techniques can also be applied to Map-Reduce programs. When applied to a Map-Reduce DAG, IDE can reuse only the output of the map layer, if the number of reducers is unchanged. MER provides more potential for re-utilization, but the map-reduce paradigm puts severe constraints on the shape of the merge function, which has to be written to perform both the reduction on the Δ and the actual merge with the previous output. However, for some applications, the reduce function could already implement the merge function (reduce needs to not change the data format and be associative).

Combining IDE and MER: Unfortunately, applying one of the two techniques reduces the potential savings of the other. However, IDE and MER can be used in conjunction by applying MER first and using IDE for the incremental DAG ($F(\Delta)$).

Related Work: The concept of incremental computation has been around for more than four decades in the programming languages community. Memoization (e.g. [14, 17, 12]) avoids re-executing functions with no side effects by caching the results of prior invocations. IDE is just an instance of memoization applied in the context of distributed data sets. The Vesta [9] software configuration management, which uses extensively memoization, provided the impetus for the current project. The dual of memoization, the dependence graph (e.g. [4, 10, 1]) tracks changes in the control graph of the program between two executions, and, based on data dependencies from new input, executes only the affected control blocks. Since this technique requires detailed visibility in the internal dynamic program state (distributed in this case), it is more complicated than our proposals.

Our work is also related to the problem of incremental view update in databases (e.g. [8, 5]). The append-only inputs assumption makes our problem easier, while the distributed nature of our computations makes the problem harder. View update solutions tend to be similar to our MER method, but they take advantage of the semantics of the database operators. More recently, continuous query techniques related to our MER method were applied to the Map-Reduce framework [13].

In this paper, we study incremental computation in the context of large-scale distributed computing. We make few assumptions about the semantics of computation and thus we use little information for performing optimizations. There is clearly a lot of interesting work to be performed in this space. We can envision a rich space of solutions built on top of our generic framework.

References

[1] ACAR, U. A., BLELLOCH, G. E., AND HARPER, R. Adaptive functional programming. In *ACM POPL* (2002).

[2] CHAIKEN, R., BOB JENKINS, P.-ØA. L., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *VLDB* (2008).

[3] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004).

[4] DEMERS, A., REPS, T., AND TEITELBAUM, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *ACM POPL* (1981).

[5] DONG, G. Incremental maintenance of recursive views: a survey. *Materialized views: techniques, implementations, and applications* (1999).

[6] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, L. The Google file system. In *SOSP* (2003).

[7] GRAY, J., AND SZALAY, A. Science in an exponential world. *Nature 440*, 23 (March 23 2006).

[8] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin* (1995).

[9] HEYDON, A., LEVIN, R., AND YU, Y. Caching function calls using precise dependencies. In *ACM SIGPLAN PLDI* (2000).

[10] HOOVER, R. Incremental graph evaluation. *PhD Thesis* (1987).

[11] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[12] LIU, Y. A., STOLLER, S. D., AND TEITELBAUM, T. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst* (1998).

[13] LOGOTHETIS, D., AND YOCUM, K. Ad-hoc data processing in the cloud. *Proc. VLDB Endow.* (2008).

[14] MICHIE, D. Memo functions and machine learning. *Nature*, 218 (1968).

[15] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD* (2008).

[16] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005), 277–298.

[17] PUGH, W., AND TEITELBAUM, T. Incremental computation via function caching. In *ACM POPL* (1989).

[18] YAHOO! Hadoop. http://hadoop.apache.org/, 2007.

[19] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008).