Nectar: Automatic Management of Data and Computation in Data Centers

Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang

Microsoft Research Silicon Valley

Abstract

Managing data and computation is at the heart of data center computing. Manual management of data can lead to data loss, wasteful consumption of storage, and laborious bookkeeping. Lack of proper management of computation can result in lost opportunities to share common computations across multiple jobs or to compute results incrementally.

Nectar is a system designed to address all the aforementioned problems. Nectar uses a novel approach that automates and unifies the management of data and computation in a data center. With Nectar, the results of a computation, called derived datasets, are uniquely identified by the program that computes it, and together with the program are automatically managed by a data center wide caching service. All computations and uses of derived datasets are controlled by the system. The system automatically regenerates a derived dataset from its program if it is determined missing. Nectar greatly improves data center management and resource utilization: obsolete or infrequently used derived datasets are automatically garbage collected, and shared common computations are computed only once and reused by others.

This paper describes the design and implementation of Nectar, and reports our evaluation of the system using both analysis of actual logs from a number of production clusters and an actual deployment on a 240-node cluster.

1 Introduction

Recent advances in distributed execution engines (MapReduce [10], Dryad [16], and Hadoop [1]) and high-level language support (Sawzall [22], Pig [21], BOOM [6], HIVE [2], SCOPE [9], DryadLINQ [26]) have greatly simplified the development of large-scale, data-intensive, distributed applications. However, major challenges still remain in realizing the full potential of data-intensive distributed computing within data centers. In current practice, a large fraction of computations in a data center are redundant and many datasets are obsolete or seldom used, wasting vast amounts of resources in a data center.

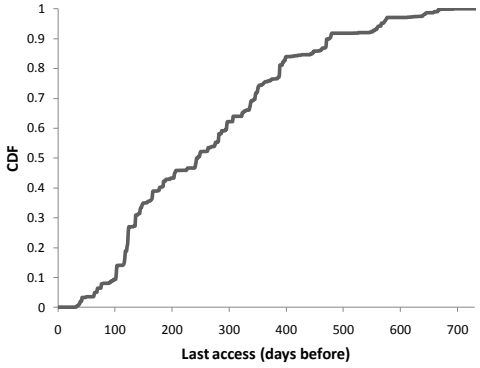

Figure 1: CDF of last access time versus amount of data in a 240-node cluster.

As one example, we quantified the wastage storage in our 240-node experimental Dryad/DryadLINQ cluster. We crawled this cluster and noted the last access time for each data file. Figure 1 shows the CDF of access time versus the amount of data. Notice that around 50% of the files are not accessed in the last 250 days.

As another example, we examined the execution statistics of 25 production clusters running data-parallel applications. We estimated that, on one such cluster, over 7000 hours of redundant computation can be eliminated per day by caching intermediate results. (This is approximately equivalent to shutting off 300 machines daily.) Cumulatively, over all clusters, this figure is over 35,000 hours per day.

Many of the resource issues in a data center arise due to lack of efficient management of either data or computation, or both. This paper describes Nectar: a system that manages the execution environment of a data center and is designed to address these problems.

Computations running on a Nectar-managed data cen-

ter are specified as programs in LINQ [3]. LINQ comprises a set of operators to manipulate datasets of .NET objects. These operators are integrated into high level .NET programming languages (e.g., C#), giving programmers direct access to .NET libraries as well traditional language constructs such as loops, classes, and modules. The datasets manipulated by LINQ can contain any .NET type, making it easy to compute with complex data such as vectors, matrices, and images. All of these operators are *functional*: they transform input datasets to new output datasets.

Data stored in a Nectar-managed data center falls in one of two classes: *primary* or *derived*. Primary datasets are created once and accessed many times. Derived datasets are the results produced by computations running on primary and other derived datasets. Examples of typical primary datasets in our data centers are click and query logs. Examples of typical derived datasets are the results of thousands of computations on those click and query logs.

In a Nectar-managed data center, all access to a derived dataset is mediated by Nectar. At the lowest level of the system, a derived dataset is referenced by the LINQ program fragment or expression that produced it. Programmers refer to derived datasets with simple pathnames that contain a simple indirection (much like a UNIX symbolic link) to the actual LINQ programs that produce them. Primary datasets are referenced by conventional pathnames.

A Nectar-managed data center offers the following four advantages.

1. Efficient space utilization. Nectar implements a cache server that manages the storage, retrieval, and eviction of the results of all computations (i.e., derived datasets). As well, Nectar retains the description of the computation that produced a derived dataset. Since programmers do not directly manage datasets, Nectar has considerable latitude in optimizing space: it can remove unused or infrequently used derived datasets and recreate them on demand by rerunning the computation. This is a classic tradeoff of storage and computation.

2. Reuse of shared sub-computations. Many applications running in the same data center share common sub-computations. Since Nectar automatically caches the results of sub-computations, they will be computed only once and reused by others. This significantly reduces redundant computations, resulting in better resource utilization.

3. Incremental computations. Many data center applications repeat the same computation on a sliding window of an incrementally augmented dataset. Again, caching in Nectar enables us to reuse the results of old data and only compute incrementally for the newly arriving data.

4. Ease of content management. With derived datasets uniquely named by LINQ expressions, and automatically managed by Nectar, there is little need for developers to manage their data manually. In particular, they don't have to be concerned about remembering the location of the data. Executing the LINQ expression that produced the data is sufficient to access the data, and incurs negligible overhead in almost all cases because of caching. This is a significant advantage because most data center applications consume a large amount of data from diverse locations and keeping track of the requisite filepath information is often a source of bugs.

Our experiments shows that Nectar, on average, could improve space utilization by at least 50%. As well, incremental and sub-computations managed by Nectar provide an average speed up of 30% for the programs running on our clusters. We provide a detailed quantitative evaluation of the first three benefits in Section 4. We have not done a detailed user study to quantify the fourth benefit, but the uniformly positive feedback from our users from our initial deployment suggests there is evidence to support our claim.

The idea of using a computational interface to describe and access data has its roots in SQL, which is ubiquitous in using queries to access database tables. Nectar takes this idea further to treat data and computation interchangably by maintaining the dependency of data and programs. It generalizes to the distributed, data center setting and handles arbitrarily complex user defined programs. Some of our ideas such as reusing results of sub-computations and caching the results of previous computations are also reminiscent of earlier work in incremental database maintenance [8], version management systems [14], and functional caching [24, 15]. Section 5 provides a more detailed analysis of our work in relation to prior research.

This paper makes the following contributions to the literature:

- We propose a novel and promising approach that automates and unifies the management of data in a data center, leading to substantial improvements in data center resource utilization.
- We present the design and implemention of our system, including a sophisticated program rewriter and static program dependency analyzer.
- We present a systematic analysis of the performance of our system from a real deployment on 240-nodes as well as analytical measurements.

The rest of this paper is organized as follows. Section 2 provides a high-level overview of the Nectar system. Section 3 describes the implementation of the system. Section 4 evaluates the system using real workloads. Section 5 covers related work and Section 6 discusses future work and concludes the paper.

2 System Design Overview

The overall Nectar architecture is shown in Figure 2. Nectar consists of a client-side component that runs on the programmer's desktop, and two services running on the data center.

Nectar is completely transparent to user programs. It uses the facilities of Dryad and DryadLINQ to manage the distribution, scheduling, and execution of the LINQ programs. Nectar takes a DryadLINQ program as input, and consults the cache service to rewrite it to an equivalent, more efficient program. It then hands the resulting program to DryadLINQ which futher compiles it into a Dryad computation running in the cluster. At run time, a Dryad job is a directed acyclic graph where vertices are programs and edges represent data channels. Vertices communicate with each other through the data channels.

Nectar makes certain assumptions about the underlying storage system. The input and output of a DryadLINQ program are expected to be *streams*. A stream consists of an ordered sequence of extents and each extent stores a sequence of object of some data type. We require that streams be append-only, meaning that new contents are added by either appending to the last extent or adding a new extent. The metadata of a stream contains Rabin fingerprints [7] of the entire stream and its extents. We use an in-house fault-tolerant, distributed file system called TidyFS that supports the necessary functionality. TidyFS, Dryad, and DryadLINQ are described in detail elsewhere [4, 16, 26] and won't be discussed further in this paper.

Nectar maintains and manages two namespaces in TidyFS. The program store keeps all DryadLINQ programs that have ever executed successfully. The data store is used to store all derived streams generated by DryadLINQ programs. The Nectar cache server provides cache hits to the program rewriter on the client side. Any stream in the data store that is not referenced by any cache entry is deemed to be garbage and deleted permanently by the Nectar garbage collector. Programs in the program store are never deleted and are used to recreate a deleted derived stream if it is needed in the future.

2.1 Client-Side Library

On the client side, Nectar takes advantage of cached results from the cache to rewrite a program P to an equivalent, more efficient program P'. It automatically inserts *AddEntry* calls at appropriate places in the program so new cache entries can be created when P' is executed. The AddEntry calls are compiled into Dryad vertices that create new cache entries at runtime. We summarize the three main client-side components below.

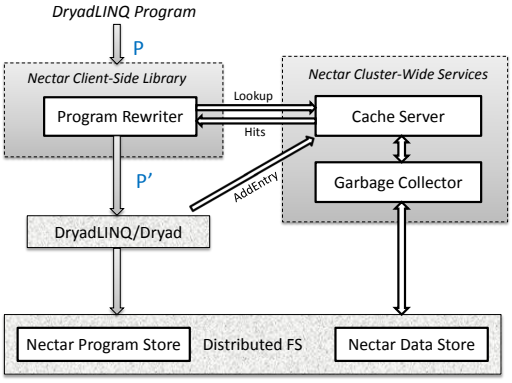

Figure 2: Nectar architecture. The system consists of a client-side library and cluster-wide service that runs in the data center. Nectar relies on the services of DryadLINQ/Dryad and TidyFS, a distributed file system.

Cache Key Calculation

A computation is uniquely identified by its program and inputs. We therefore use the Rabin fingerprint [7] of the program and the input datasets as the cache key for a computation. The input datasets are stored in TidyFS and their fingerprints are calculated based on the actual stream contents. Nectar calculates the fingerprint of the program and combines it with the fingerprints of the input datasets to form the cache key.

The fingerprint of a DryadLINQ program must be able to detect any changes to the code the program depends on. However, the fingerprint should not change when code the program does not depend on changes. This is crucial for the correctness and practicality of Nectar. (Fingerprints can collide but the probability of a collision can be made vanishingly small by choosing long enough fingerprints.) We implement a static dependency analyzer to compute the transitive closure of all the code that can be reached from the program. The fingerprint is then formed using all the reachable code. Of course, our analyzer only produces an over-approximation of the true depdendency.

Rewriter

Nectar rewrites user programs to use cached results where possible. Specifically, we support the following three rewriting scenarios that arise in practice.

Common sub-expressions. Internally, a DryadLINQ program is represented as a LINQ expression tree. Nectar treats all prefix sub-expressions of the expression tree as candidates for caching and looks up in the cache for possible cache hits for every prefix sub-expression.

Incremental query plans. Incremental computation on datasets is a common occurence in data intensive computing. Typically, a user has run a program P on input D. Now, he is about to compute P on input $D + D'$, the concatenation of D and D'. The Nectar rewriter finds a new operator to combine the results of computing on the old input and the new input separately. That is, it finds an operator C such that $P(D + D') = C(P(D), D')$. Nectar automatically derives C for most operators in LINQ.

Incremental query plans for sliding windows. This is the case where data is appended to the end of the input of a repeated computation while the beginning of the input is excluded from the computation. That is, the same program is repeatedly run on the following sequence of inputs:

$$
\begin{aligned}
D_1 &= d_1 + d_2 + ... + d_n, \\
D_2 &= d_2 + d_3 + ... + d_{n+1}, \\
D_3 &= d_3 + d_4 + ... + d_{n+2}, \\
&.....
\end{aligned}
$$

Nectar automatically generates cache entries for each individual dataset d_i, and uses them in subsequent computations.

In the real world, a program may belong to more than one category above. For example, an application that analyzes logs of the past seven days is rewritten as an incremental computation by Nectar, but Nectar may use sub-expression results of log preprocessing on each day from other applications.

Cost Estimator

An expression might hit different entries in the cache server with different sub-expressions and/or partial input datasets. So there are typically multiple alternatives to choose from in rewriting a DryadLINQ program. The rewriter uses a cost estimator to choose an optimal one from multiple alternatives.

Comparing the cost of two candidates is not always straightforward. For example, it is hard to determine if a shorter prefix with a larger input dataset is better than a longer prefix with a smaller input dataset. Nectar estimates the cost of alternative candidates generated by rewriter using execution statistics collected and saved in the cache server from past executions. We discuss the details of the cost estimation in Section 3.1.

2.2 Datacenter-Wide Service

The datacenter-wide service in Nectar comprises two separate components: the cache service and the garbage collection service. The actual datasets are stored in the distributed storage system and the datacenter-wide services manipulate the actual datasets by maintaining pointers to them.

Cache Service

Nectar implements a distributed datacenter-wide cache service for bookkeeping information about DryadLINQ programs and the location of their results. The cache service has two main functionalities: (1) serving the cache lookup requests by the Nectar rewriter; and (2) managing derived datasets by deleting the cache entries of the least value.

Programs of all successful computations are uploaded to a dedicated program store in the cluster. Thus, the service has the necessary information about cached results, meaning that it has a recipe to recreate any derived dataset in the data center. When a derived dataset is deleted but needed in the future, Nectar recreates it using the program that produced it. If the inputs to that program have themselves been deleted, it backtracks recursively till it hits the immutable primary datasets or cached derived datasets. Because of this ability to recreate datasets, the cache server can make informed decisions to implement a cache replacement policy, keeping the cached results that yield the most hits and deleting the cached results of the least value when storage space is low.

Garbage collector

The Nectar garbage collector operates transparently to the users of the cluster. Its main job is to identify datasets unreachable from any cache entry and delete them. We use a standard mark-and-sweep collector. Actual content deletion is done in the background without interfering with the concurrent activities of the cache server and job executions. To avoid the races with concurrent creation of new deriveds, we use a lease to protect newly created derived datasets.

3 Implementation Details

This section presents the implementation details of Nectar. We focus on the most important aspects of the system. Section 3.1 describes how computation caching is achieved. Section 3.2 discribes how the derived datasets are managed automatically.

3.1 Caching Computations

Nectar rewrites a DryadLINQ program to an equivalent but more efficient one using cached results. This generally involves: 1) identifying all sub-expressions of the expression, 2) probing the cache server for all cache hits for the sub-expressions, 3) using the cache hits to rewrite it into a set of equivalent expressions, and 4) choosing one that gives us the maximum benefit based on some cost estimation.

Cache and Programs

A cache entry records the result of executing a program on some given input. It is of the form:

$$\langle FP_{PD}, FP_P, Result, Statistics, FPList \rangle$$

Here, FP_{PD} is the combined fingerprint of the program and its input datasets, FP_P is the fingerprint of the program only, $Result$ is the location of the output, and $Statistics$ contains execution and usage information of this entry. The last field $FPList$ contains a list of fingerprint pairs each representing the fingerprints of the first and last extents of an input dataset. As we shall see later, it is used by the rewriter to efficiently search the solution space. Since the same program could be executed on different inputs, there can be multiple cache entries with the same FP_P.

We use FP_{PD} as the primary key. So our caching is sound only if FP_{PD} can uniquely determine the result of the computation. The fingerprint of the inputs is based on the actual content of the datasets. For a large dataset, the fingerprint is formed by combining the fingerprints of its extents which are efficiently computed in parallel in the data center.

The computation of the program fingerprint is tricky, as the program may contain user-defined functions that call into library code. We implemented a static dependency analyzer to capture all the dependency of an expression. At the time a DryadLINQ program is invoked, DryadLINQ knows all the dynamic linked libraries (DLLs) it depends on. We divide them into two categories: system and application. For a system DLL, we assume it is available and identical on all cluster machines and therefore is not included in the dependency. For an application DLL that is written in native code (e.g., C or assembler), we include the entire DLL as a dependency. For an application DLL that is in managed code (e.g., C#), our analyzer traverses the call graph to compute all the code reachable from the initial expression.

The analyzer works at the bytecode level. It uses standard .NET reflection to get the body of a method, finds all the methods being called in the body, and traverses to those methods recursively. When a virtual method call is encountered, we include all the possible call sites. While our analysis is certainly a conservative approximation of the true dependency, it is quite precise and works well in practice. Since dynamic code generation could introduce unsoundness into the analysis, it is forbidden in managed application DLLs, and is statically enforced by the analyzer.

The statistics information kept in the cache entry is used by the rewriter to find an *optimal* execution plan. It is also used to implement the cache insertion and eviction policy. It contains information such as the *cumulative execution time*, the number of hits on this entry, and the last access time. The cumulative execution time is defined as the sum of the execution time of upstream Dryad vertices of the current execution stage. It is computed at the time of the cache entry insertion using Dryad logging information.

The cache server supports a very simple client interface. The important operations include: (1) `Lookup(fp)` finds and returns the cache entry for the given primary key `fp`; (2) `Inquire(fp)` returns all cache entries that have `fp` as their FP_P; and (3) `AddEntry` inserts a new cache entry. Note that the argument of `Lookup` is FP_{PD}. We will see their use in the following sections.

The Rewriting Algorithm

Having explained the structure and interface of the cache, let us now look at how Nectar rewrites a program.

For a given expression, we may get cache hits on any possible sub-expression and subset of the input dataset, and considering all of them in the rewriting is not tractable. We therefore only consider cache hits on prefix sub-expression on segments of the input dataset. More concretely, consider a simple example `D.Where(P).Select(F)`. The `Where` operator applies a filter to the input dataset `D`, and the `Select` operator applies a transformation to each item in the input. Let us assume that the input `D` has n extents. We will only consider cache hits for the sub-expressions `S.Where(P)` and `S.Where(P).Select(F)` for all subsequence of extents in `D`.

Our rewriting algorithm is a simple recursive procedure. We start from the largest prefix sub-expression, the root of the expression. Below is an outline of the algorithm:

Step 1. At each sub-expression, we probe the cache server to obtain all the possible hits on it. There can be multiple hits on different subsequences of the input `D`. Let us denote the set of hits by H.

Step 2. If there is a hit on the entire `D`, we just use that hit and stop exploring its sub-expressions, because it

gives us the most saving in terms of cumulative execution time. Otherwise, we compute the best hit for the current expression using smaller prefixes, and then choose the best among it and H. To do that, we recursively apply our procedure on each successor of the current expression and find the best hits for all of them, which is combined to form the candidate solution.

Step 3. Now, there are $|H|+1$ candidates to rewrite the current expression: The $|H|$ hits from Step 1 and a new one by combining all the hits of the smaller prefixes from Step 2. Our job now is to choose a subset of it such that they operate on disjoint subsequence of `D` and give us the most saving in terms of cumulative execution time. This boils down to the well-known problem of computing the maximum independent sets of an interval graph, which has a known efficient solution using dynamic programming techniques [11].

In Step 1, the rewriter calls `Inquire` to compute H. As described before, `Inquire` returns all the possible cache hits of the program with different inputs. A real hit means that its input dataset is identical to a subsequence of extents of `D`. A bruteforce search is inefficient and requires to check every subsequence. As an optimization, we store in the cache entry the fingerprints of the first and last extents of the input dataset. With that information, we can compute H in linear time.

The main step of rewriting a program P on incremental data is to derive a combining operator C such that $P(D + D') = C(P(D), D')$, where C combines the results of applying P separately on the datasets D and D'. Nectar supports all the LINQ operators DryadLINQ supports.

The combining functions for some LINQ operators require the parallel merging of multiple streams, and are not directly supported by DryadLINQ. We introduced three combining functions `MergeSort`, `HashMergeGroups`, and `SortMergeGroups`, which are straightforward to implement using DryadLINQ's Apply operator [26]. `MergeSort` takes multiple sorted input streams, and merge sorts them. HashMergeGroups and SortMergeGroups take multiple input streams and merge groups of the same key from the input streams. If all the input streams are sorted, Nectar chooses to use SortMergeGroups, which is streaming and more efficient. Otherwise, Nectar uses HashMergeGroups. We give an example later in this section.

Cache Insertion Policy

We consider every prefix sub-expression of an expression to be a candidate for caching. Adding a cache entry incurs additional cost if the entry is not useful. It requires us to store the result of the computation on disk (instead of possibly pipelining the result to the next stage), incurring the additional disk IO and space overhead. Obviously it is not practical to cache everything. Nectar implements a simple strategy to determine what to cache.

First of all, Nectar always creates a cache entry for the final result of a computation as we get it for free: it does not involve a break of the computation pipeline and incurs no extra IO and space overhead.

For sub-expression candidates, we wish to cache them only when they are predicted to be useful in the future. However, determining the potential usefulness of a cache entry is generally difficult. So we base our cache insertion policy on some simple, intuitive heuristics. The caching decision is made in the following two phases.

First, when the rewriter rewrites the expression, it decides on the places in the expression to insert AddEntry calls. This is done using the usage statistics maintained by the cache server. The cache server keeps some simple statistics for a sub-expression based on request history from clients. In particular, it records (1) the number of times it has been looked up, and (2) the number of times we have got cache hits on this expression. (There can be multiple cache entries for the same expression with different inputs.) On response to a cache lookup, these two numbers are included in the return value. We insert an AddEntry call only when both the number of lookups and cache hits exceed pre-defined thresholds.

Second, the decision made by the rewriter may still be wrong because of the lack of information about the saving of the computation. Information such as execution time and disk consumption are only available at run time. So the final insertion decision is made based on the runtime information of the execution of the sub-expression. Currently, it is a simple benefit function that is proportional to the execution time and inversely proportional to storage overhead. We add the cache entry when the benefit exceeds a threshold.

We also make our cache insertion policy adaptive to storage space pressure. When there is no pressure, we choose to cache more aggressively as long as it saves machine time. This strategy could increase the useless cache entries in the cache. But it is not a problem because it is addressed by Nectar's garbage collection, which we will discuss in more detail in Section 3.2.

Example: GroupBy-Select

GroupBy-Select performs a *MapReduce*-type job and is one of the most important computation patterns for data-parallel computation. We now use it as a concrete example to illustrate Nectar caching.

GroupBy-Select is expressed in DryadLINQ as follows:

```
var groups = source.GroupBy(KeySelect);
```

```
var reduced = groups.Select(Reduce);
```

The input of `GroupBy` is a sequence of records. It first groups the records into groups using the keys computed by the function `keySelect`, and then applies the reduction function `Reduce` to each group. In a distributed setting, `GroupBy` first partitions the records based on their keys across a cluster of machines, and then forms the groups and applies `Reduce` to each group independently on each partition in parallel.

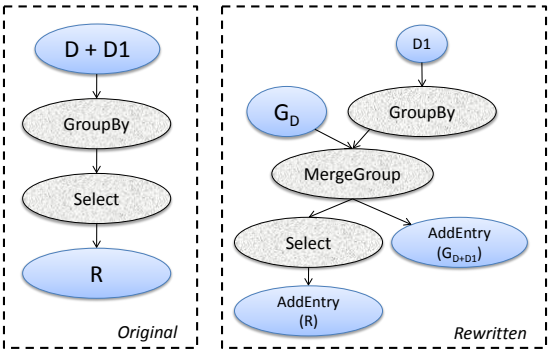

Figure 3: Rewriting of GroupBy(D+D1)

Let us look at the interesting case of incremental computation. Figure 3 shows the rewriting of GroupBy-Select in the presence of caching. Let us assume we have performed the same computation on input D and added a cache entry for the result of `GroupBy`. So when we encounter the same computation with input $D + D_1$, the Nectar rewriter would get a cache hit on G_D. So it only needs to perform `GroupBy` on D_1 and merge with G_D to form new groups. We compute `GroupBy` on D_1 the same way as G_D, generating the same number of partitions with the same partition scheme. We then do a pairwise merge with G_D to construct the result dataset G_{D+D_1}. This allows the system to reuse the partitioning and ordering properties of G_D for G_{D+D_1}. We always create a cache entry for the final result. It is also important to create a cache entry for the new groups (G_{D+D_1}), because it will be useful when the same computation is performed on $G_{D+D_1+D_2}$ in the future.

Similar to MapReduce's combiner optimization [10], DryadLINQ can decompose `Reduce` into the composition of two associatve and commutative functions if `Reduce` is determined to be decomposable. We handle this by first applying the decomposition as in [25] and then the caching and rewriting as described above.

3.2 Managing Derived Data

Derived datasets can take up a significant amount of storage space in a data center, and a large portion of it could be unused or seldom used. Nectar keeps track of the usage statistics of all derived datasets and deletes the ones of the least value. Recall that Nectar permanently stores the program of every derived dataset so that a deleted derived can be recreated by re-running its program.

Data Store for Derived Data

As mentioned before, Nectar stores all derived datasets in a data store inside a distributed, fault-tolerand file system. The actual location of a derived dataset is completely opaque to programmers. Accessing an existing derived dataset must go through the cache server. We expose a simple, standard file interface with one important restriction: New derived datasets can only be created as results of computations.

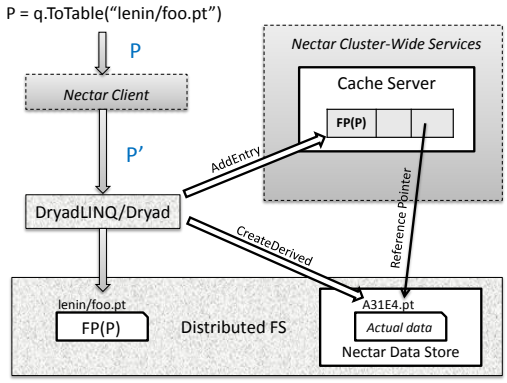

Figure 4: The creation of a derived dataset. The actual dataset is stored in the Nectar data store. The user file contains only the primary key of the cache entry associated with the derived.

Our scheme to achieve this is quite simple. Figure 4 shows the flow of creating a derived dataset by a computation and the relationship between the user file and the actual derived dataset. In the Figure, `P` is a user program that writes its output to `lenin/foo.pt`. After applying tranformations by Nectar and DryadLINQ, it is executed in the data center by Dryad. When the execution succeeds, the actual derived dataset is stored in the data store with a unique name generated by Nectar. A cache entry is created with the fingerprint of the program (`FP(P)`) as the primary key and the unique name as a field. The content of `lenin/foo.pt` just contains the primary key of the cache entry.

To access `lenin/foo.pt`, Nectar simply uses `FP(P)` to look up the cache to obtain the location of the actual derived dataset (`A31E4.pt`). The fact that all accesses go through the cache server allows us to keep track of the usage history of every derived dataset and to implement automatic garbage collection for deriveds based on their usage history.

Garbage Collection

When the available disk space falls below a threshold, the system automatically deletes the derived datasets that are considered to be least useful in the future. This is achieved by a combination of the Nectar cache server and garbage collector.

A derived dataset is protected from garbage collection if it is referenced in any cache entry. So, the first step is to inform the cache server to delete cache entries that it determines to have the least value. The datasets referred to by these deleted cache entries will then be considered garbage and collected by the Nectar garbage collector.

The information we store in the cache entries allows us to make informed decisions on the usefulness of the cache entries. Our eviction policy is based on the *cost-to-benefit ratio*. Suppose S is the size of the derived dataset referred to by a cache entry and ΔT is the time interval between now and the time it was last used. Let us also assume that N is the number of times the cache entry is used and M is the cumulative machine time of the computation that created this cache entry. The cost-to-benefit ratio of a cache entry is then defined as

$$\text{Ratio} = (S \times \Delta T)/(N \times M)$$

When a garbage collection is triggered, Nectar scans the entire cache, computing the cost-to-benefit ratio for each cache entry. It then sorts the cache entries according to the ratios and deletes the top n entries such that the collective space saving reaches a pre-defined threshold. This entire operation is done in the background, concurrently with any other cache server operations. Since we don't have enough information to compute a useful cost/benefit ratio for them, we exclude newly created cache entries to give them a chance to demonstrate their usefulness. A lease on each cache entry prevents its deletion until the lease expires.

When the cache server completes its eviction of cache entries, the garbage collector starts to delete all derived datasets that are not protected by any existing cache entry. We use a simple mark-and-sweep collector. Again, this is done in the background, concurrently with any other activities in the system.

Operations such as a Dryad job can run currently with the garbage collector and create new cache entries and derived datasets. Derived datasets pointed to by cache entries (freshly created or otherwise) are not candidates for garbage collection. Notice however that freshly created derived datasets, which due to concurrency may not yet have a cache entry, also need to protected from garbage collection. We do this with a lease on the dataset.

With these leases in place, garbage collection is quite straightforward. We first compute the set of all derived datasets (ignoring the ones with unexpired leases) in our data store, exclude from it the set of all derived datasets referenced by cache entries, and treat the remaining as garbage.

Our garbage collection and cache eviction could mistakenly delete datasets that are subsequently requested, but these can be recreated by rexecuting the appropriate program(s) from the program store.

Programs are stored in binary form in the program store. A program is a Dryad job that is ready to submit to the data center for execution. In particular, it includes the execution plan and all the application DLLs. We exclude all system DLLs, assuming that they are available on the data center machines. For a typical data center that runs 1000 jobs daily, our experience suggests it would take less than 1TB to store one year's program in uncompressed form. With compression, it should take up roughly a few hundreds of gigabytes of disk space, which is negligible even for a small sized data center.

4 Experimental Evaluation

We evaluate Nectar running on our 240-node research cluster. We also present the results from our analysis of detailed execution logs from 25 large production clusters that run data-intensive parallel jobs similar to the ones on our research cluster. We first present our analytic results.

4.1 Production Clusters

We use logs from 25 different clusters to evaluate the usefulness of Nectar. The logs consists of detailed execution statistics for jobs in these clusters for a recent 3-month period. Across these clusters, 33182 jobs were executed in the given period. For each job in a cluster, the log has the source program and detailed execution statistics such as computation time, bytes read and written and the actual time taken for every stage in a job. The log also gives information on the submission time, start time, end time, user information, and job status.

Programs from the production cluster work with massive datasets such as click logs and search logs. Programs are written in a language similar to DryadLinq in that each program is a sequence of SQL-like queries. A program is compiled into an expression tree with various stages and modeled as a DAG with vertices representing processes and edges representing data flows. The DAGs are executed on a Dryad cluster, just as in Nectar managed Dryad/DryadLINQ cluster. Input data in these clusters is stored as append-only streams. Most data streams are partitioned into stream sets based on date.

4.1.1 Benefits from Caching

We parse the execution logs to recreate a set of DAGs, one for each job. The root of the DAG represents the

input to the job and a path through the DAG starting at the root represents a partial (i.e., a sub-) computation of the job. Two paths from different DAGs that are identical represents an opportunity to save part of the computation time of the later job by caching results from the first.We simulate the effect of Nectar's caching on these DAGs to estimate both sub-computation and incremental/sliding window cache hits.

Our results show that 20% to 65% jobs in a cluster benefits from caching. In fact, 30% of the jobs in 17 clusters had a cache hit, and on an average more than 35% of the jobs benefited from caching.

The log contains detailed computation time information for each node in the DAG for a job. When there is a cache hit on a sub-computation of a job, we can therefore calculate the time saved by the cache hit.

Figure 5 shows that significant percentage of computation time can be saved in each cluster with Nectar. Most clusters can save a minimum of 20% to 40% of computation time and in some clusters the savings are up to 50%.

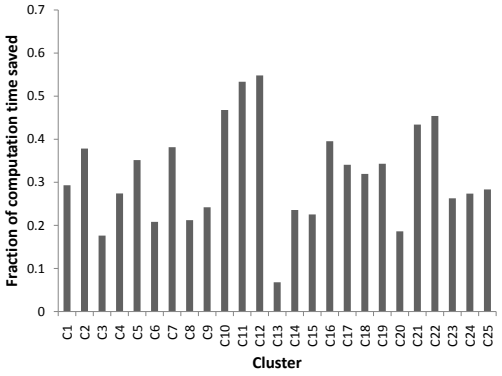

Figure 5: Fraction of compute time saved in each cluster

Table 1 shows the minimum hours of computation time that can be saved per day in each cluster. These numbers show significant savings, for instance, a minimum of 7143 hours of computation per day can be saved using Nectar in Cluster C5. This is roughly equivalent to saying that about 300 machines in that cluster were doing wasteful computations all day that caching could eliminate. Across all 25 clusters, 35078 hours of computation per day can be saved.

4.1.2 Ease of Program Development

Our analysis of the caching accounted for both sub-computation as well as incremental/sliding window hits. We noticed that, the percentage of sliding window hits in some production clusters was minimal (under 5%). We investigated this further and noticed that many programmers explicitly structure their programs so that they can reuse a previous computation. This somewhat artificial structure makes their programs cumbersome, which can be alleviated by using Nectar.

Cluster	Computation Time Savings (hours/day)	Cluster	Computation Time Savings (hours/day)
C1	3898	C14	753
C2	2276	C15	755
C3	977	C16	2259
C4	1345	C17	3385
C5	7143	C18	528
C6	62	C19	4
C7	57	C20	415
C8	590	C21	606
C9	763	C22	2002
C10	2457	C23	1316
C11	1924	C24	291
C12	368	C25	58
C13	105		

Table 1: Minimum Computation Time Savings

There are anecdotes of system administrators manually running a common sub-expression on the daily input and explicitly notifying programmers to avoid each program performing the computation on its own and tying up cluster resources. Nectar automatically supports incremental computation and programmers do not need to code them explicitly. As discussed in Section 2, Nectar tries to produce the best possible query plan using the cached results significantly reducing computation time, at the same time making it opaque to the user.

An unanticipated benefit of Nectar reported by our users on the research cluster was that it aids in debugging during program development. Programmers incrementally test and debug pieces of their code. With Nectar the debugging time significantly improved due to cache hits. We therefore try to quantify the effect of this on the production clusters. We assumed that a program is a debugged version of another program if they had almost the same queries accessing the same source streams and writing the same derived streams, submitted by the same user and had the same program name.

Table 2 shows the amount of debugging time that can be saved by Nectar in the 90 day period. We present results for the first 12 clusters due to space constraints. Again, these are conservative estimates but shows substantial savings. For instance, in Cluster C1, a minimum of 3 hours of debugging time can be saved per day. Notice that this is real elapsed time, i.e., each day 3 hours of computation on the cluster spent on debugging programs can be avoided with Nectar.

Cluster	Debugging Time Saved (hours)	Cluster	Debugging Time Saved (hours)
C1	270	C7	3
C2	211	C8	35
C3	24	C9	84
C4	101	C10	183
C5	94	C11	121
C6	8	C12	49

Table 2: Actual elapsed time saved on debugging in 90 days.

4.1.3 Managing Storage

Today, in data centers, storage is manually managed.[1] We studied storage in our 240-node research cluster that has been used by a significant number of users over the last 2 to 3 years. As we pointed out in Section 1, we crawled this 240-node cluster for derived objects and noted their last access times. 109 TB of derived datasets were created in the last 2 years. Figure 1 shows the CDF of the amount of derived data and their access time. As we see, about 50% of the data (54.5 TB) was never accessed in the last 250 days. This shows that users often create derived datasets and after a point, forget about them, leaving them occupying unnecessary storage space.

We analyzed the production logs for the amount of derived datasets written. When calculating the storage occupied by these datasets, we assumed that if a new job writes to the same dataset as an old job, the dataset is overwritten. Figure 6 shows the growth of derived data storage in cluster C1. It show an approximatley linear growth with the total storage occupied by datasets created in 90 days being 670 TB.

Cluster	Projected unreferenced derived data (in TB)
C1	2712
C5	368
C8	863
C13	995
C15	210

Table 3: Projected unreferenced data in 5 production clusters

Assuming that the trend on data access times in our local cluster is similar on the production cluster, Table 3 shows the projected space occupied by unreferenced derived datasets in 5 production clusters that showed linear growth of data similar to cluster C1. Any object that has not been referenced in 250 days is deemed unreferenced. This result is obtained by extrapolating the amount of data written by jobs in 90 days to 2 years based on the storage growth curve and predicting that 50% of that storage is not accessed in the last 250 days (based on the result from our local cluster). As we see, production clusters create large amount of derived datasets and if not properly managed can create significant storage pressure.

[1]Nectar's motivation in automatically managing storage partly stems from the fact that we used to get periodic e-mail messages from the administrators of the production clusters requesting us to delete our derived objects to ease storage pressure in the cluster.

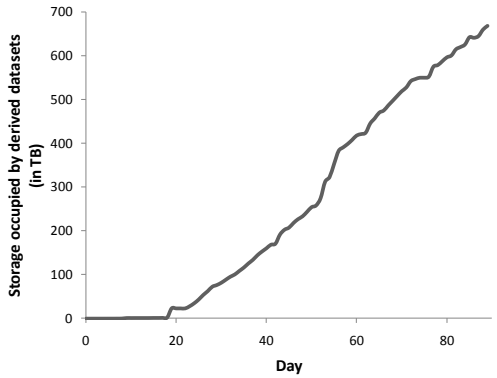

Figure 6: Growth of storage occupied by derived datasets in Cluster C1

4.2 System Deployment Experience

Each machine in our 240-node research cluster has two dual-core 2.6GHz AMD Opteron 2218 HE CPUs, 16GB RAM, four 750GB SATA drives, and runs Windows Server 2003 operating system. We evaluate the comparative performance of several programs that run on our cluster with Nectar turned on and off. The programs were written by researchers in our lab.

4.2.1 Datasets

We use three datasets to evaluate performance of Nectar.

WordDoc Dataset. The first dataset is a collection of Web documents. Each document record contains a document URL and its content (as a list of words). The data size is 987.4 GB . The dataset is randomly partitioned into 236 partitions. Each partition has two replicas in the distributed file system, evenly distributed on 240 machines.

ClickLog Dataset. The second dataset is a small sample of about 160GB collected over five consecutive days from an anonymized click log of a commercial search engine. The dataset is randomly partitioned into 800 partitions, two replicas each, evenly distributed on 240 machines.

SkyServer Dataset. This database is taken from the Sloan Digital Sky Survey database [12]. It contains two

data files: 11.8 and 41.8 GBytes of data. Both files were manually range-partitioned into 40 partitions using the same keys.

4.2.2 Sub-computation Evaluation

We have four programs: *WordAnalysis*, *TopWord*, *MostDoc*, and *TopWordRatio* that analyze the *WordDoc* dataset.

WordAnalysis parses the dataset to generate the number of occurrences of each word and the number of documents that it appears in. *TopWord* looks for the top ten most commonly used words in all documents. *MostDoc* looks for the top ten words appearing in the largest number of documents. *TopWordRatio* finds the percentage of occurrences of the top ten mostly used word among all words. All programs take the entire 987.4 GB dataset as input.

Program Name	Cumulative Time		Saving
	Nectar on	Nectar off	
TopWord	16.1m	21h44m	98.8%
MostDoc	17.5m	21h46m	98.6%
TopRatio	21.2m	43h30m	99.2%

Table 4: Saving by sharing a common sub-computation: Document analysis

With Nectar on, we can cache the results of executing the first program, which spends a huge amount of computation analyzing the list of documents to output an aggregated result of much smaller size (12.7 GB). The subsequent three programs share a sub-computation with the first program, which is satisfied from the cache. Table 4 shows the cumulative CPU time saved for the three programs. This behavior is not isolated, one of the programs that uses the *ClickLog* dataset shows a similar pattern; we don't report the results here for reasons of space.

4.2.3 Incremental Computation

We describe the performance of a program that studies query relevance by processing the *ClickLog* dataset. When users search a phrase at a search engine, they click the the most relevant URLs returned in the search results. Monitoring the URLs that are clicked the most for each search phrase is important to understand query relevance. This program is an example where the initial dataset is large, but the incremental updates are small. The input to the query relevance program is the set of all click logs collected so far, which increases each day, because a new log is appended daily to the dataset.

Table 5 shows the cumulative CPU time with Nectar on and off, the size of datasets and incremental updates each day. We see that the total size of input data increases each day, while the computation resource used daily increases much slower when Nectar is on. We observed similar performance results for another program that calculates the number of active users, who are those that clicked at least one search result in the past three days. These results are not reported here for reasons of space.

	Data Size(GB)		Time (m)		Saving
	Total	Update	On	Off	
Day3	68.20	40.50	93.0	107.5	13.49%
Day4	111.25	43.05	112.9	194.0	41.80%
Day5	152.19	40.94	164.6	325.8	49.66%

Table 5: Cumulative machine time savings for incremental computation.

4.2.4 Debugging Experience: Sky Server

Here we demonstrate how Nectar saves program development time by shortening the debugging cyle. We select the most timeconsuming query (Q18) from the Sloan Digital Sky Survey database [12]. The query identifies a gravitational lens effect by comparing the locations and colors of stars in a large astronomical table, using a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. The query is composed of four steps, each of which is debugged separately. When debugging the query, the first step failed and the programmer modified the code. Within a couple of tries, the first step succeeded, and execution continued to the second step, which failed, and so on.

Table 6 shows the average savings in cumulative time in one round of debugging for each step with Nectar. Towards the end of the program, Nectar saves as much 94% of the time.

	Cumulative Time		Saving
	Nectar on	Nectar off	
Step1	47.4m	47.4m	100%
Step2	26.5m	125.0m	79.80%
Step3	35.5m	245.5m	85.54%
Step4	15.0m	258.7m	94.20%

Table 6: Debugging: SkyServer cumulative time

5 Related Work

In term of the overall system architecture, we drew inspiration from the Vesta system [14]. Many high-level concepts and techniques such as the clear separation of primary and derived data are directly taken from Vesta. However, because of the difference in application domains, the actual design and implementation of the main

system components such as caching and program rewriting are radically different.

With the wide adoption of distributed execution platforms like Dryad/DryadLINQ, MapReduce/Sawzall, Hadoop/Pig [16, 26, 10, 22, 1, 21], recent work has investigated job patterns and resource utilization in data centers [23, 20, 5, 19, 13]. These investigation of real work loads have revealed a vast amount of wastage in data centers due to redundant computations, which is consistent with our findings from logs of a number of production clusters.

DryadInc [23] represented our early attempt to eliminate redundant computations via caching, even before we started on the DryadLINQ project. The caching approach is quite similar to Nectar. However, it works at the level of Dryad dataflow graph, which is too general and too low-level for the system we wanted to build.

The two systems that are most related to Nectar are the stateful bulk processing system [19] and Comet [13]. The systems mainly focus on addressing the important problem of incremental computation, which is also one of the problems Nectar is designed to address. However, Nectar is a much more ambitious system, attempting to provide a comprehensive solution to the problem of automatic management of data and computation in a data center.

As a design principle, Nectar is designed to be transparent to the users. The stateful bulk processing system takes a different approach by introducing new primitives and hence makes *state* explicitly in the programming model. It would be interesting to understand the tradeoffs in terms of performance and ease of programming.

Comet, also built on top of Dryad and DryadLINQ, also attempted to address the sub-computation problem by co-scheduling multiple programs with common subcomputations to execute together. There are two interesting issues raised by the paper. First, when multiple programs are involved in caching, it is difficult to determine if two code segments from different programs are identical. This is particularly hard in the presence of user-defined functions, which is very common in the kind of DryadLINQ programs targeted by both Comet and Nectar. It is unclear how it is achieved in Comet. Nectar addresses this problem by building a sophisticated static program analyzer that allows us to compute the dependency of user-defined code. Second, co-scheduling in Comet requires submissions of multiple programs with the same timestamp. It is therefore not useful in all scenarios. Nectar instead shares sub-computations across multiple jobs executed at different times by using a datacenter-wide, persistent cache service.

The caching aspect of our work is closely related to the incremental view maintenance in databases [8, 17]. In incremental data management, they study the problem of updating the materialized views incrementally when their base tables are updated. Nectar is simpler in that we only consider append-only updates, while databases attempt to handle random updates to the base table. On the other hand, Nectar is more challenging because we must deal with user-defined functions written in a general-purpose programming language.

Caching function calls in a functional programming language is well studied in the literature [24, 18, 14]. Memoization avoids re-computing the same function calls by caching the result of past invocations. Caching in Nectar can be viewed as function caching in the context of large-scale distributed computing.

6 Discussion and Conclusions

In this paper, we described Nectar, a system that automates the management of data and computation in data centers. The system has been deployed on a 240-node research cluster, and has been in use by a small number of developers. Feedback has been quite positive. The most popular comment from our users is that the system makes program debugging much more interactive and fun. Most of us, the Nectar developers, use Nectar to develop Nectar on a daily basis, and found a big increase in our productivity.

To validate the effectiveness of Nectar, we performed a systematic analysis of computation logs from 25 production clusters. As reported in Section 4, we have seen huge potential value in using Nectar to manage the computation and data in a large data center. Our next step is to work on transfering Nectar to Microsoft production data centers.

Nectar is a complex distributed systems with multiple interacting policies. Devising the right policies and fine-tuning their parameters to find the righ tradeoffs are essential to make the system work in practice. Our evaluation of these tradeoffs has been limited, but we are actively working on this top. We hope we will continue to learn a great deal with the ongoing deployment of Nectar on our 240-node research cluster.

What Nectar essentially does is to unify computation and data, treating them interchangably by maintaining the dependency between them. This allows us to greatly improve the data center management and resource utilization. We believe this is a very powerful paradigm and represents a significant step forward in data center computing.

References

[1] The Hadoop project. `http://hadoop.apache.org/`.

[2] The HIVE project. `http://hadoop.apache.org/hive/`.

[3] The LINQ project. `http://msdn.microsoft.com/netframework/future/linq/`.

[4] Tidyfs. `http://research.microsoft.com/en-us/projects/tidyfs/`.

[5] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. In *Proceedings of VLDB Endowment*, 2008.

[6] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. BOOM: Data-centric programming in the datacenter. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[7] A. Z. Broder. Some applications of rabins fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag, 1993.

[8] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 1991.

[9] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *International Conference of Very Large Data Bases (VLDB)*, August 2008.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[11] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[12] J. Gray, A. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. Vandenberg. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting*, pages 189–210, Paris, France, March 2002. Carleton Scientific. also as MSR-TR-2002-01.

[13] 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 *ACM Symposium on Cloud Computing (SOCC)*, 2010.

[14] A. Heydon, R. Levin, T. Mann, and Y. Yu. *Software Configuration Management Using Vesta*. Springer-Verlag, 2006.

[15] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of Programming language design and implementation (PLDI)*, 2000.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2007.

[17] K. Y. Lee, J. H. Son, and M. H. Kim. Efficient incremental view maintenance in data warehouses. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, 2001.

[18] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 1998.

[19] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Stateful bulk processing for incremental algorithms. In *ACM Symposium on Cloud Computing (SOCC)*, 2010.

[20] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (Industrial Track) (SIGMOD)*, Vancouver, Canada, June 2008.

[22] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[23] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, San Diego, CA, June 15 2009.

[24] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the*

Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL), 1989.

[25] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.