1

Large-scale Machine Learning using DryadLINQ

Mihai Budiu, Dennis Fetterly, Michael Isard,
Frank McSherry, and Yuan Yu

This chapter describes DryadLINQ, a general-purpose system for large scale data-parallel computing, and illustrates its use on a number of machine learning problems.

The main motivation behind the development of DryadLINQ was to make it easier for non-specialists to write general purpose, scalable programs that can operate on very large input datasets. In order to appeal to non-specialists we designed the programming interface to use a high level of abstraction that insulates the programmer from most of the detail and complexity of parallel and distributed execution. In order to support general-purpose computing we embedded these high-level abstractions in .NET, giving developers access to full-featured programming languages with rich type systems and proven mechanisms (such as classes and libraries) for managing complex, long-lived and geographically distributed software projects. In order to support scalability over very large data and compute clusters the DryadLINQ compiler generates code for the Dryad runtime, a well-tested and highly efficient distributed execution engine.

As machine learning moves into the industrial mainstream and operates over diverse data types including documents, images and graphs, it is increasingly appealing to move away from domain-specific languages like Matlab and towards general-purpose languages that support rich types and standardized libraries. The examples in this chapter demonstrate that a general-purpose language such as C# supports effective, concise implementations of standard machine learning algorithms, and that DryadLINQ efficiently scales these implementations to operate over hundreds of computers and very large datasets primarily limited by disk capacity.

1.1 Manipulating datasets with LINQ

We use Language Integrated Queries, or LINQ (Microsoft, 2010), as our programming model for writing large-scale machine learning applications. LINQ adds high level declarative data manipulation to many of the .NET programming languages, including C#, Visual Basic and F#. This section provides a short introduction to LINQ.

LINQ comprises a set of operators to manipulate collections of .NET objects. The operators are integrated seamlessly in high level .NET programming languages, giving developers direct access to all the .NET libraries as well the traditional language constructs such as loops, classes, and modules. The collections manipulated by LINQ operators can contain any .NET type, making it easy to compute with complex data such as vectors, matrices, and images. As it will be shown in the rest of this chapter, many machine learning algorithms can be naturally and elegantly expressed using LINQ.

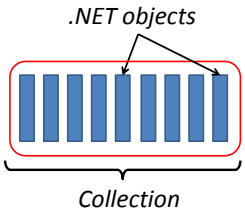

Figure 1.1 LINQ data model: collections of typed values.

LINQ datasets are .NET collections. Technically, a .NET collection of values of type `T` is a data type which implements the predefined interface `IEnumerable<T>`. Many commonly used data structures such as arrays, lists, hash-tables, and sets are such collections. The elements of a collection can be any type, including nested collections. Figure 1.1 illustrates the abstract LINQ data model. We will see later that this model can be naturally extended to accommodate very large collections that span multiple computers. The `IEnumerable` interface provides access to an *iterator*, used to enumerate the elements of the collection. Programmers can use these iterators to scan over the data sets.

To simplify programming, LINQ provides a large set of operators to manipulate collections, drawn from common data parallel programming patterns. All of these operators are *functional*: they transform input collections to completely new output collections, rather than update the existing collections in place. Although there are many primitive LINQ operators (and the users can easily add more), all of them can be seen as variants of the 7 oper-

ators listed in Table 1.1. Readers familiar with the SQL database language will find these operators quite natural.

Operation	Meaning
`Where`	(Filter) Keep all values satisfying a given property.
`Select`	(Map) Apply a transformation to each value in the collection.
`Aggregate`	(Fold, Reduce) Combine all values in the collection to produce a single result (e.g., max).
`GroupBy`	Create a collection of collections, where the elements in each inner collection all have a common property (*key*).
`OrderBy`	(Sort) Order the elements in the collection according to some property (*key*).
`SelectMany`	(Flatten) Generates a collection for each element in the input (by applying a function), then concatenates the resulting collections.
`Join`	Combine the values from two collections when they have a common property.

Table 1.1 *Essential LINQ operators.*

Most LINQ operators take as a parameter at least one *function* used to process the elements in the collection. These functions are most commonly *anonymous functions*, a convenient .NET shorthand written as `x => f(x)` for the function mapping a single variable x to a result $f(x)$. The anonymous function bodies can invoke user defined methods, or may simply consist of primitive .NET operations. For example, the anonymous function `x => x%2` computes the value of the input argument modulo 2. Anonymous functions with multiple inputs are written by parenthesizing the inputs together, as in `(x,y,z) => f(x,y,z)` for the case of three inputs.

To be concrete, table 1.2 shows the result of applying some LINQ operators to the collection `C` = (1,2,3,4,5). The only example that may not be self-explanatory in Table 1.2 is `Join`, the only operation that we have shown that operates on two collections. `Join` receives three function arguments: (1) the first function argument (in our example `x=>x`) computes a *key* value for each element in the left collection; (2) the second function (`x=>x-4`) computes the key value for each element in the right collection; (3) finally, the third function `(x,y)=>x+y` reduces pairs of values, where `x` is from the first collection and `y` from the second collection. This function is invoked only for pairs of values that have matching keys. In our example, the only matching pair of values is 1 and 5, whose keys are both 1 (1 and repectively 5-4), and thus the result of the `Join` is a collection with a single element 1+5.

The final feature of LINQ we introduce is the `IQueryable<T>` interface, deriving from the `IEnumerable<T>` interface. An object of type `IQueryable<T>`

Operation	Result
`C.Where(x => x > 3)`	(4,5)
`C.Select(x => x + 1)`	(2,3,4,5,6)
`C.Aggregate( (x,y) => x+y)`	15
`C.GroupBy(x => x % 2)`	((1,3,5), (2,4))
`C.OrderBy(x => -x)`	(5,4,3,2,1)
`C.Select(x => Factors(x))`	((1), (1, 2), (1, 3), (1, 2, 4), (1, 5))
`C.SelectMany(x => Factors(x))`	(1, 1, 2, 1, 3, 1, 2, 4, 1, 5)
`C.Join(C, x=>x, x=>x-4, (x, y)=>x+y)`	(6)

Table 1.2 *Examples using LINQ operators on collection* `C`=*{1,2,3,4,5}*. `Factors` *is a user defined function.*

represents a *query* (i.e., a computation) that can produce a collection with elements of type `T`. The queries are not evaluated until an element or aggregate is required from the collection[1]. Applying LINQ operators to an `IQueryable` object produces a new `IQueryable` object, describing the computation required to produce the new result.

Importantly, each `IQueryable<T>` can specify a *LINQ provider*, capable of examining the query and choosing from many different execution strategies. Many LINQ providers exist: PLINQ (Duffy, 2007) executes queries on a single computer using multiple CPU cores, LINQ to SQL translates LINQ queries to SQL statements executed on a database engine. DryadLINQ (Yu et al., 2008) itself is simply a LINQ provider which executes the queries on a computer cluster.

1.2 k-means in LINQ

We now show how to use LINQ to implement a basic machine-learning algorithm; in Section 1.3.4 we will show how this program can be executed in a distributed fashion. k-means is a classical clustering algorithm which divides a collection of vectors into k clusters. The clusters are represented by their centroids; each vector belongs to the cluster with the nearest centroid. This is an iterative computation, which is performed until a termination criterion is reached.

LINQ collections can contain arbitrary types, and for our purposes we use a class `Vector` providing all the usual vector arithmetic operations (addition, scalar product, dot product, L2 norm, etc.). The `Vector` class could be pre-

[1] Queries are a form of *lazy evaluation* of code; this is encountered in other programming languages such as Haskell or Scheme.

defined and imported from some shared library. We can then represent a collection of vectors using `IQueryable<Vector>`.

We first define a useful auxiliary function `NearestCenter` that computes the nearest neighbor of a vector from a set of vectors.

```
Vector NearestCenter(Vector point, IQueryable<Vector> centers)
{
    var nearest = centers.First();
    foreach (var center in centers)
        if ((point - center).Norm() < (point - nearest).Norm())
            nearest = center;

    return nearest;
}
```

The k-means algorithm is a simple iterative computation: each iteration groups the input vectors by their nearest center, and averages each group to form the centers for the next iteration. The `KMeansStep` function below computes the updated centers from the input vectors and current centers. The LINQ code simply groups the input vectors using the nearest center as a key, and uses aggregation to reduce each group to its average:

```
IQueryable<Vector> KMeansStep(IQueryable<Vector> vectors,
                              IQueryable<Vector> centers)
{
   return vectors.GroupBy(vector => NearestCenter(vector, centers))
                 .Select(g => g.Aggregate((x,y) => x+y) / g.Count());
}
```

The k-means algorithm repeatedly invokes this step until a termination condition is met. The example below uses a fixed number of iterations, though more complex convergence criteria could be employed.

```
IQueryable<Vector> KMeans(IQueryable<Vector> vectors,
                          IQueryable<Vector> centers,
                          int iterations)
{
    for (int i = 0; i < iterations; i++)
        centers = KMeansStep(vectors, centers);

    return centers;
}
```

The result of the `KMeans` function is a single object with type `IQueryable<Vector>`, describing the computation necessary to produce the result from `iterations` steps of our iterative algorithm. Only when the user attempts to enumerate the result of `KMeans` will the query be executed and the iterations performed.

1.3 Running LINQ on a cluster with DryadLINQ

In order to perform computations on very large datasets we need to pool the resources of multiple computers. Fortunately, the computations expressed in LINQ are very easy to parallelize by distributing work to multiple computers. The software stack that we have built for this purpose is shown in Figure 1.2. In this text we particularly focus on two layers of this stack: Dryad and DryadLINQ. Layers such as Cluster storage and Cluster services, which provide a distributed file system and execution of processes on cluster machines, are important, but are outside the scope of this chapter.

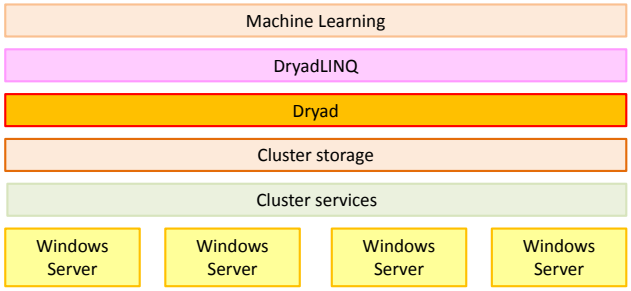

Figure 1.2 Software stack for executing LINQ programs on a cluster of computers.

1.3.1 Dryad

Dryad (Isard et al., 2007) is a software layer that coordinates the execution of multiple dependent programs (processes) running on a computer cluster. A Dryad job is a collection of processes that communicate with each other through uni-directional *channels*. Dryad allows the programmer to describe the computation as a directed acyclic multigraph, in which nodes represent processes and edges represent communication channels. The requirement that the graphs be acyclic may seem restrictive, but it enables Dryad to provide automatically fault-tolerance, without any knowledge of the application semantics. Moreover, we will see that many interesting algorithms can be expressed as acyclic graphs. Figure 1.3 shows a hypothetical example of a Dryad execution plan.

Dryad handles the reliable execution of the graph on a cluster. Dryad schedules computations to computers, monitors their execution, collects and reports statistics, and handles transient failures in the cluster by re-executing

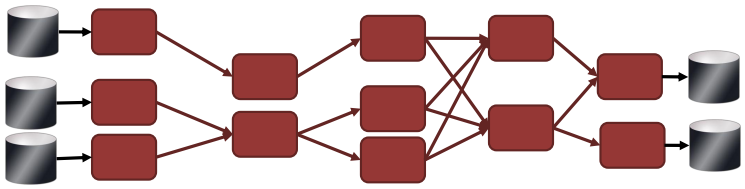

Figure 1.3 Example of a hypothetical Dryad job execution plan; the nodes are programs that execute, possibly on different computers, while the edges are channels transporting data between the processes. The input and output of the computation reside on the cluster storage medium..

failed or slow computations. Dryad jobs execute in a *shared-nothing* environment: there is no implicit shared memory or disk state between the various processes in a Dryad job; the only communication medium between processes are the channels themselves.

1.3.2 DryadLINQ

We have introduced two essential ingredients for implementing large-scale cluster computation: a parallel language (LINQ) and an execution environment for clusters (Dryad). We now describe DryadLINQ, a compiler and runtime library that bridges the gap between these two layers. DryadLINQ translates programs written in LINQ into Dryad job execution plans that can be executed on a cluster by Dryad, and transparently returns the results to the host application.

DryadLINQ presents the same data model as LINQ to the programmers. But, in order to distribute the computation across multiple computers, DryadLINQ internally partitions the data into disjoint parts, as shown in Figure 1.4. The original collections become collections of partitions; the partitions being some (smaller) LINQ collections that reside on individual computers. (The partitions residing on the cluster storage medium can optionally be replicated on several computers each, for increased fault-tolerance.)

DryadLINQ implements LINQ operators over partitioned collections. Figure 1.5 shows how this is done for some of the basic LINQ operators from Table 1.1. Operators like `Select`, `SelectMany` and `Where` are the easiest to implement, since they operate on individual elements; they can be applied to individual parts regardless of the partitioning scheme. The `GroupBy` requires

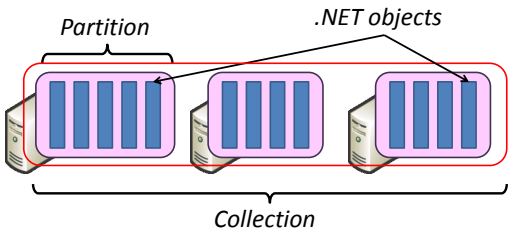

Figure 1.4 DryadLINQ data model: collections of typed values partitioned among several computers. Compare with Figure 1.1.

records with the same key to be co-located, so it is implemented in two steps: (1) repartition the collection using a deterministic hash function applied to the grouping key; (2) after repartitioning all elements with the same key are present on the same computer, which can perform a standard LINQ `GroupBy` on the local data to produce the necessary collection of groups. Aggregation using an associative function can be done hierarchically: in a first phase the data in each part is aggregated independently, in subsequent phases subsets of intermediate results are combined, until in the last phase a single computer performs the final aggregation.

Figure 1.6 shows the translation for two of LINQ operators that generate binary collection operations. The first example results from the nested usage of collections (when an inner collection is used for all elements in the outer collection, as we will see in Section 1.3.4): in the generated Dryad graph the inner collection is broadcast to all partitions of the outer collection.

The second example shows an implementation of the binary `Join` operator. Similar to `GroupBy` it is implemented using deterministic hash function, ensuring that elements with matching keys end up in corresponding partitions.

The Dryad job execution plans generated by DryadLINQ are *composable*: the output of one graph can become the input of another one. In fact, this is exactly how complex LINQ queries are translated: each operator is translated to a graph independently, and the graphs are then concatenated. The graph generation phase is followed by a graph rewriting phase that performs optimizations, and which can substantially alter the shape of the job execution plan. As a simple example, sequences of `Select` and `Where` operations can be pipelined and executed within a single vertex.

In general, during the computation, the collection elements must be moved between computers, so the in-memory data structures need to be *serialized* to a shared physical medium, either a disk or the network. DryadLINQ exploits its full knowledge of the types in the collections to automatically

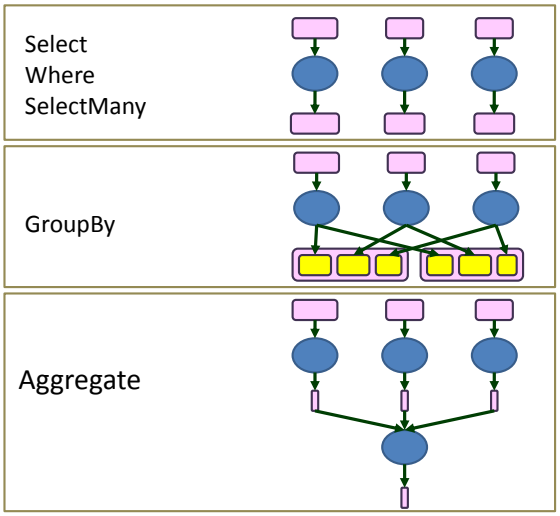

Figure 1.5 Dryad jobs generated by DryadLINQ for the simplest LINQ operators.

generate efficient serialization and de-serialization code. The user can always replace the default serialization routines with custom ones, but this is seldom needed. DryadLINQ also optionally compresses data before writing it to disk or transmitting it across the network.

1.3.3 Map-Reduce and DryadLINQ

Any Map-Reduce (Dean and Ghemawat, 2004) program can be easily translated into a DryadLINQ program. In consequence, any algorithm expressed using the Map-Reduce framework can be also implemented in DryadLINQ. The Map-Reduce approach requires the programmer to specify "map" and "reduce" functions, where the map function transforms each input record to a list of keyed intermediate records, and the reduce function transforms a group of intermediate records with the same key into a list of output records[2].

```
IQueryable<R> MapReduce<S,T,R,K>(
       IQueryable<S> records,
```

[2] Map-Reduce as defined by Google specifies that a reducer will receive the records sorted on their keys; in our implementation each reducer is only given all the records that have the same key. DryadLINQ is flexible enough to emulate the exact behavior of Map-Reduce as well, but we omit this implementation for simplicity.

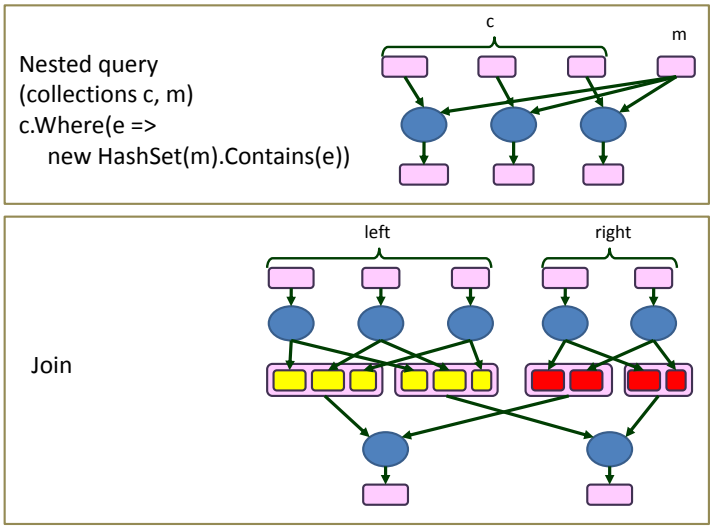

Figure 1.6 Dryad jobs generated by DryadLINQ for other LINQ operators.

```
        Func<S,IEnumerable<KeyValuePair<K,T>> mapper,
        Func<IGrouping<K,T>,IEnumerable<R>> reducer)
{
    return records.SelectMany(mapper)
                  .GroupBy(temp => temp.Key, temp => temp.Value)
                  .SelectMany(reducer);
}
```

There are some simple but noteworthy observations about using LINQ and DryadLINQ to implement Map-Reduce:

The LINQ version of Map-Reduce is strongly typed (the type of the elements in the input and output is known at compilation time), so more errors are caught at compilation time (this feature becomes very useful once programs become large). LINQ also provides complete integration with .NET libraries and existing integrated development environments; this immediately leverages the effort put in to reusable libraries and development tools. Finally, as LINQ supports many providers, the computation can be immediately executed across a variety of LINQ providers: multicore PLINQ, LINQ to SQL, and DryadLINQ.

When using DryadLINQ, in particular, a few additional advantages emerge: Due to strong typing, DryadLINQ can generate very efficient serialization code for all objects involved, without the need to resort writing to manual serialization code, such as Protocol Buffers (Google, 2010). By using

DryadLINQ to execute the Map-Reduce programs we inherit all of DryadLINQ's optimizations: computations are placed close to the data, multiple Map-Reduce programs can be composed, and optimizations can be applied across the Map-Reduce boundaries. Map-Reduce computations can even be mixed in with other LINQ computations that are difficult to express in Map-Reduce (for example, Joins). Finally, the eager aggregation performed by DryadLINQ discussed in Section 1.3.4 is a generalization of the concept of *combiners* and *reducers* that Map-Reduce uses, but DryadLINQ can automatically infer the combiners and reducers in many cases (Yu et al., 2009).

1.3.4 k-means Clustering in DryadLINQ

The power of DryadLINQ is illustrated by how little the k-means program from Section 1.2 needs to change to be executed on a cluster of computers. To invoke DryadLINQ we only need to change the input collection of a query to be one of the partitioned collections shown in Figure 1.4.

While using DryadLINQ is easy for the programmer, under the hood many optimizations concur to provide an efficient execution of the queries. Recall the core of our k-means iteration:

```
IQueryable<Vector> KMeansStep(IQueryable<Vector> vectors,
                              IQueryable<Vector> centers)
{
   return vectors.GroupBy(vector => NearestCenter(vector, centers))
                 .Select(g => g.Aggregate((x,y) => x+y) / g.Count());
}
```

The `GroupBy` operation at the heart of the k-means aggregation collects a very large amount of data; even if the input vectors are initially spread over hundreds of balanced partitions, if half of them belong to a single cluster it would seem that the runtime would need to bring them to a single computer in order to compute the average. (This is the problem of *data skew*, which is notoriously difficult to handle in a generic way.) Such a strategy would severely overload the computer computing the centroid for the large group. However, the DryadLINQ optimizer uses a robust *eager aggregation* algorithm to implement this particular computation(Yu et al., 2009). By inspecting the code for the centroid computation, DryadLINQ can infer that the computation of the average is associative and commutative. DryadLINQ thus generates a job execution plan that uses two-level aggregation (similar to the plan shown at the bottom Figure 1.5): each computer builds local groups with the local data and only sends the aggregated information about these groups to the next stage; the next stage computes the actual centroid.

DryadLINQ can often determine automatically whether a computation is associative and commutative; when this is unfeasible, the user can employ the C# annotation mechanism to tag functions. For example, we tagged the vector addition operation with the `[Associative]` attribute for this optimization to work in our case (not shown). Figure 1.7 shows the plan that is generated for this program.

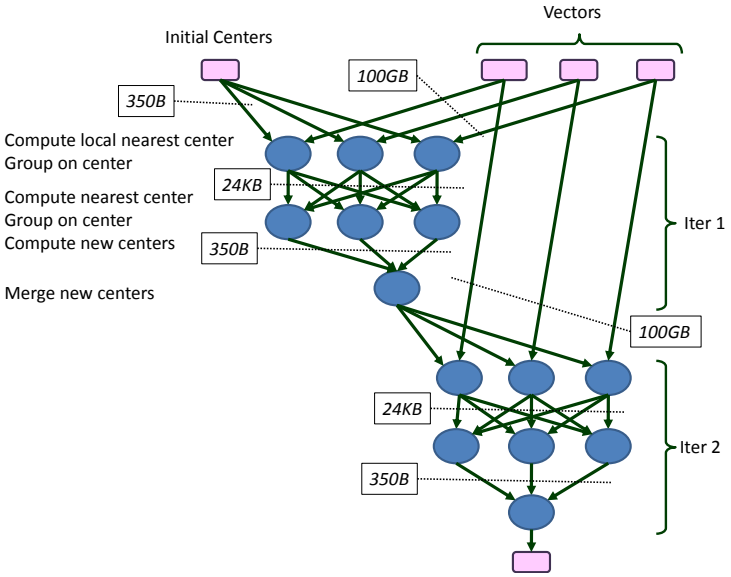

Figure 1.7 Dryad job execution plan generated for two iterations of the k-means algorithm on ten dimensional vectors, with $k = 10$. The vector data is split into three partitions. The boxes with dotted lines show the amount of data exchanged between stages for a 100GB set of vectors.

The key selection function for the `GroupBy` operation uses the centers from the previous iteration, using the nested pattern from Figure 1.6. DryadLINQ produces a plan that updates the centers once, and broadcasts the results once to each part, where they are re-used. This optimization also allows us to chain multiple iterations of k-means together, without interrupting the computation on the cluster. This reduces the overhead for launching jobs on the cluster and allows DryadLINQ to optimize execution across iteration boundaries.

Measurements

For our measurements we use a collection of random vectors with 10 dimensions whose total size is 100GB. Each vertex computes k pre-aggregated cluster centers, each exactly 10 doubles (one per dimension), which are then

exchanged, aggregated, and re-broadcast to each of the vertices in the next stage in the following iteration, independent of the number or distribution of vectors on each computer. The main bottleneck in data-parallel computations tends to be the data exchange, where the shared network fabric must support many point-to-point data transfers. The local operations are limited by the speed of reading data from the local disks, and do only modest processing. We are thus presenting measurements just for the amount of data exchanged across the network. Figure 1.7 shows the amount of data read by each stage; the output of the first stage is only 24KB (we have used 31 partitions in this particular execution). The majority of the time is spent in the first stage of each iteration (computing local centers).

1.3.5 Decision Tree Induction in DryadLINQ

For our next DryadLINQ example, we consider the problem of computing a decision tree. We use a binary decision tree to classify records with the following structure:

```
class Record
{
    bool label;        // class the record belongs to
    bool[] attributes; // attributes to classify on
}
```

A decision tree is a tree of attribute choices, terminating in leaves with class labels on them. The tree is used to classify records by starting from the root, examining a specified attribute at each internal node, proceeding down the branch indicated by the attribute's value, and continuing recursively until a leaf (and class label) is reached. We will represent a decision tree with a dictionary that maps tree node indices (integer values) to attribute indices in the `attribute` array: given a node index `node` in the tree, `tree[node]` is an index in the `attributes` array, indicating which attribute is tested by the node.

```
// compute index of node in (partial) tree reached by a record
int TreeWalk(Record record, Dictionary<int, int> tree)
{
    var node = 0;
    while (tree.ContainsKey(node))
        node = 2 * node + (record.attributes[tree[node]] ? 1 : 2);

    return node;
}
```

The most common algorithm to induce a decision tree starts from an

empty tree and a set of records with class labels and attributes with values. The algorithm repeatedly extends the tree by grouping records by their current location under the partial tree, and for each such group determining the attribute resulting in the greatest reduction in conditional entropy (of the class label given the attribute value). For example, we might write:

```
records.GroupBy(record => TreeWalk(record, tree))
       .Select(group => FindBestAttribute(group));
```

While this approach makes perfect sense in a single-computer setting, in the data-parallel setting it has the defect that all of the input records must be reshuffled in each iteration. Moreover, single computers can be overloaded when many records map to a single node in the tree (for example, during the first few levels of the tree) — the data skew issue discussed in Section 1.3.4.

Instead, we consider an alternate "bottom up" algorithm with a highly parallel execution plan. We use (but do not show here) a function `CondEntropy` computing the conditional entropy of a list of lists of counts.

```
IEnumerable<Pair<int, int>>
DecisionTreeLayer(IQueryable<Record> data, Dictionary<int, int> tree)
{
    // emit a 4-tuple for each attribute,
    var a = data.SelectMany(x => x.attrs.Select((y, i) => new
                                        {
                                            prefix = TreeWalk(x, tree),
                                            label = x.record.label,
                                            index = i,
                                            value = y
                                        }));

    // count unique quadruples
    var b = a.GroupBy(x => x)
             .Select(g => new { g.Key, count = g.Count() });

    // compute conditional entropy for each attribute in each prefix
    var c = b.GroupBy(x => new { x.Key.prefix, x.Key.index })
             .Select(x => new
             {
                 x.Key.prefix,
                 x.Key.index,
                 entropy = CondEntropy(x.GroupBy(x => x.value))
             });

    // group by prefix, return min-entropy attribute
    return c.GroupBy(x => x.prefix)
            .Select(g => g.OrderByDescending(y => y.entropy).First())
            .Select(x => new Pair<int, int>(x.prefix, x.index));
}
```

The computation proceeds in four steps:

1. The first step replaces each record with a collection of quadruples, one for each of the record's attributes. The quadruple contains the record's location in the current tree, the record's class label, the index of the corresponding attribute, and the attribute's value.
2. The second step aggregates all identical quadruples, counting the number of occurrences of each and performing the most significant data reduction in the computation.
3. The third step groups the counts from the second step, using the pair (tree prefix, attribute index) as the key and then computes the entropy of these groups (which is the conditional entropy of this attribute).
4. Finally, the fourth step performs another grouping on the set identifier, selecting the attribute with the lowest conditional entropy (by using the `OrderBy` LINQ operator to sort the attributes and using the `First` LINQ operator to choose the one with minimum entropy). The result of this computation is list of set identifiers and optimal attribute index for each. This list can be used to attach a new layer of nodes to the decision tree.

The code presented computes a new level in the decision tree. To compute a full tree, we would write:

```
var records = PartitionedTable.Get<Record>(datafile);

var tree = new Dictionary<int, int>();
for (int i = 0; i < maxTreeDepth; i++)
    foreach (var result in DecisionTreeLayer(records, tree))
        tree.Add(result.Key, result.Value);
```

Each iteration through the loop invokes a query returning the list of attribute indices that are best for each of the leaves in the old tree that we aimed to extend. In principle, we could unroll the loop to a single DryadLINQ computation as we did with the k-means computation, using an `IQueryable<Pair<int,int>>` as the data structure for our tree, and simply feeding the result of one layer in as the tree for the next, but we do not do this here. Instead, the `tree` variable is updated on the client computer, and retransmitted to the cluster by DryadLINQ with each iteration.

The plan generated for the decision tree layer is shown in Figure 1.8. One can view this plan as a sequence of map-reduce computations; in the resulting plan each "reduce" stage is fused with the following "map" stage. This plan also fundamentally benefits from DryadLINQ's eager aggregation; before any data exchange happens, each part is reduced to a collection of

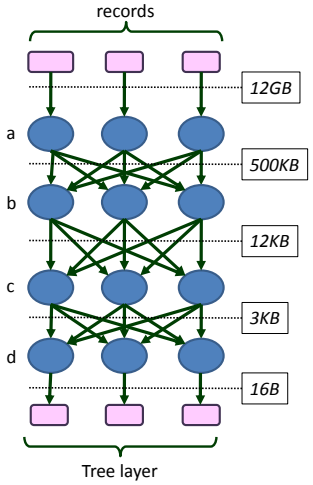

Figure 1.8 Dryad job execution plan generated for computing one layer of the decision tree, assuming that the records data is split into three partitions. The dotted lines show the amount of data that is crossing between layers when computing the second level of the tree for a 12GB input set.

counts, no more than the $|sets| \times |labels| \times |attributes| \times |values|$. The number of records plays no role in the size of the aggregates. As the tree becomes deeper, the number of sets will increase, and there may come a point where it is more efficient to reshuffle the records rather than their resulting aggregates. However, the number of aggregates never exceeds the number of quadruples, which never exceeds the number of attributes present in the records.

Measurements

As for the k-means algorithm, the volume of data transferred across the network by the decision tree induction code is largely independent on the volume of training data. Each group results in a number of aggregates bounded by the structure of the problem, rather than the number or distribution of records. We might see fewer aggregates if the records are concentrated properly (e.g., clustered by label, so that each part only produces half of the possible aggregates), but the performance on random data is a good worst-case evaluation.

We have used a 12GB input dataset for these measurements. Figure 1.8 shows the amount of data that crosses between computation stages; the second stage reads only 0.5MB, due to the local aggregation performed by

DryadLINQ. The amount of data written by the last stage doubles for each successive tree layer computation.

1.3.6 Example: Singular Value Decomposition

The Singular Value Decomposition (SVD) lies at the heart of several large scale data analyses: principal components analysis, collaborative filtering, image segmentation, and latent semantic indexing, among many others. The SVD of a $n \times m$ matrix A is a decomposition $A = U\Sigma V^T$ such that U and V are both orthonormal ($U^TU = V^TV = I$) and Σ is a diagonal matrix with non-negative entries.

Orthogonal Iteration is a common approach to computing the U and V matrices, in which candidates U and V are repeatedly updated to AV and A^TU, respectively, followed by re-orthonormalization of their columns. In fact, only one of the two iterates need to be retained (we will keep V) as the other can be recomputed with one step. The process converges in the limit to the true factors, and convergence after any fixed number of iterations can be quite good; the error is exponentially small in the number of iterations, where the base of the exponent depends on the conditioning of the matrix.

We will represent a matrix as a collection of `Entry` objects, commonly reserved for sparse matrices but not overly inefficient for dense matrices.

```
struct Entry
{
    int row, col
    double val;
}
```

Based on this representation we can write several standard linear algebra operations using LINQ operations:

```
// aggregates matrix entries with the same coordinates into a single value
IQueryable<Entry> Canonicalize(IQueryable<Entry> a)
{
    return a.GroupBy(x => new { x.row, x.col }, x => x.val)
            .Select(g => new Entry(g.Key.row, g.Key.col, g.Sum()));
}

// multiplies matrices. best if one is pre-partitioned by join key
IQueryable<Entry> Multiply(IQueryable<Entry> a, IQueryable<Entry> b)
{
    return Canonicalize(a.Join(b,
        x => x.col,
        y => y.row,
        (x, y) => new Entry(x.row, y.col, x.val * y.val)));
}
```

```
IQueryable<Entry> Add(IQueryable<Entry> a, IQueryable<Entry> b)
{
    return Canonicalize(a.Concat(b));
}

IQueryable<Entry> Transpose(IQueryable<Entry> a)
{
    return a.Select(x => new Entry(x.col, x.row, x.val));
}
```

`Multiply` produces a substantial amount of intermediate data, but DryadLINQ's eager aggregation significantly reduces this volume before the data are exchanged across the network.

These operations are sufficient for us to repeatedly compute A^TAV, but they do not let us orthonormalize the columns of V. However, the $k \times k$ matrix V^TV is quite small, and contains enough information to produce (via Cholesky decomposition) a $k \times k$ matrix L_V so that VL_V is orthonormal. We will use DryadLINQ to compute V^TV and return this value to the client computer where we compute L_V and introduce it into the computation.

The orthogonal iteration algorithm then looks like:

```
// Cholesky decomposition done on the local computer (not shown)
PartitionedTable<Entry> Cholesky(IQueryable<Entry> vtv);

// materialize a and a^T partitioned on columns
var a = a.HashPartition(x => x.col).ToPartitionedTable("a");
var at = Transpose(a).HashPartition(x => x.col)
                     .ToPartitionedTable<Entry>("at");

// run 100 orthogonal iteration steps
for (int iteration=0; iteration < 100; iteration++)
{
    v = Multiply(at, Multiply(a, v));

    // Perform Cholesky decomposition once every five iterations
    if (iteration % 5 == 0)
    {
        v = v.ToPartitionedTable("svd-" + iteration.ToString());
        v = Multiply(v, Cholesky(Multiply(Transpose(v), v)));
    }
}
```

Although it can also be written as a LINQ program, the body of the `Cholesky` function is not shown; it is executed on the client computer. On each loop iteration DryadLINQ creates a query that "wraps around" the `for` loop, computing essentially $A^T \times A \times (V \times L_V)$. The orthonormalization

step is only required for numerical stability and is executed only once every 5 iterations. A new DryadLINQ job is created and dispatched to the cluster once for every 5 iterations of the loop. Figure 1.9 shows the shape of the DryadLINQ job execution plan generated for this program.

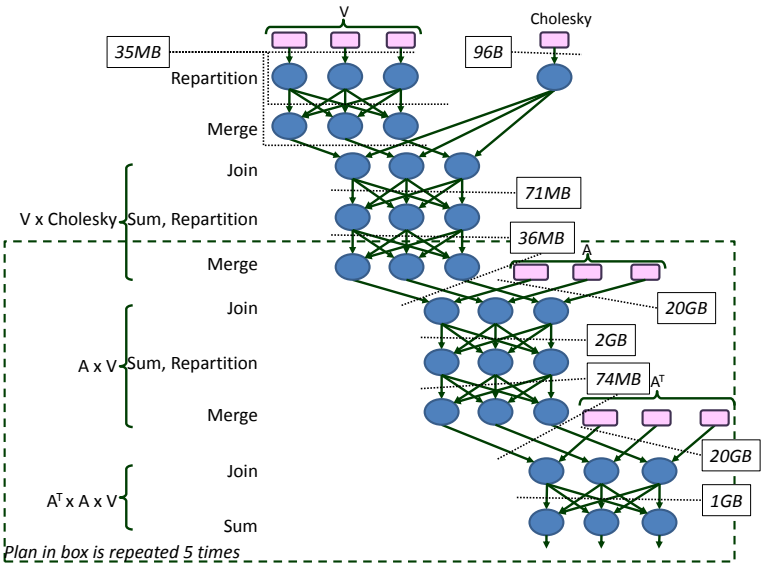

Figure 1.9 Partial Dryad job execution plan generated for the SVD computation, assuming that the matrices V and A are split into three partitions. The portion of the plan in the dotted box is repeated 4 more times. The dotted lines show the volume of data between computation stages for a 20GB A matrix.

Each loop iteration involves a `Join` of V with A, and with A^T. We use the `HashPartition` DryadLINQ-specific operator (an extension to basic LINQ) to give a hint to the system to pre-partition A using its columns as keys; as a consequence the join against the rows of V does not move any of A's entries across the network; only entries corresponding to V, usually much smaller, are moved. Likewise, we keep a copy of A^T partitioned by its columns. Although keeping multiple copies of A may seem wasteful, the cost is paid in terms of cheap disk storage rather than a scarce resource like memory.

Measurements

As we have noted, to extract optimum performance from the SVD algorithm it is important to pre-partition the matrix data by row and column, avoiding full data exchanges in each iteration. As such, matrix structure can play a large role in the performance of the algorithm: matrices with block

structure, partitioned accordingly, result in substantially fewer aggregates than matrices partitioned randomly. We evaluate our SVD algorithm on a random sparse matrix; we used a matrix A of 20GB. The figure 1.9 shows the volume of data that is crossing between stages; because A is rectangular, multiplication with A or A^T generates a different amount of intermediate data. Without the local aggregation feature of DryadLINQ the result of a `Join` would be 72GB, the actual data exchanged in our implementation is 2GB. The final multiplication result is even smaller, at 74MB.

1.4 Lessons Learned

We have applied DryadLINQ to a large variety of data mining and machine learning problems, including: decision trees, neural networks, linear regression, expectation maximization, probabilistic latent semantic indexing, probabilistic index maps, graphical models, principal component analysis. We summarize here some of the lessons we have learned in this process.

1.4.1 Strengths

The main strength of DryadLINQ is the very powerful high-level language which integrates into a single source program both single-computer and cluster-level execution. The seamless transition between the two environments allows one to build easily very complex applications using just Visual Studio as a tool for development and debugging.

When necessary, interoperation with other languages (and in particular with native code) is easily achieved using the standard .NET mechanisms for invoking unmanaged code. We sometimes have to rely on native code either for speed or for legacy reasons.

When writing very large programs the richness of the datatypes manipulated by DryadLINQ and the strong typing of the language are particularly helpful. The strong typing enables DryadLINQ to automatically generate all the code for serializing the data moved between computers. For some projects the amount of serialization code can dwarf the actual analysis.

Since the output of DryadLINQ is also LINQ code, but running on individual partitions, we have been able to make use of other existing LINQ providers, such as PLINQ, which parallelizes the application across multiple cores, using effectively all the computers in the cluster.

1.4.2 Weaknesses

While DryadLINQ is a great tool to program clusters, there is a price to pay too for the convenience that it provides. We discuss here several weaknesses that we have identified.

Efficiency: managed code (C#) is not always as efficient as native code (C++); in particular, arithmetic and string operations can be up to twice as fast in native code.

Debugging: debugging problems that occur when processing large data sets is not always easy. DryadLINQ provides some good tools for debugging, but the experience of debugging a cluster program remains more painful than debugging a single-computer program.

Transparency: finally, while DryadLINQ does provide a high level language to write cluster applications, one cannot just hide behind the language abstraction and hope to get efficient code. In most cases one needs to have some understanding of the operation of the compiler and particularly of the job execution plans generated (this is why we have shown the job execution plans for all our examples); this knowledge enables one to avoid egregious mistakes and to choose the queries that exhibit the best performance.

1.4.3 A Real Application

As an industrial-strength application of DryadLINQ, we have implemented (in collaboration with other researchers) several machine learning projects for Microsoft's Xbox Project Kinect. The goal of Kinect is to provide a natural interface to the Xbox gaming console, by tracking the users' bodies and voices in real time; this transforms the user's body itself into a game controller. The visual input device for the Kinect system is a combination video+depth camera (measuring the color and distance to each pixel in the image), operating in real time at video frame rate. The output of the Kinect system, available to the application developers, is the 3D position of the body joints (a skeleton) of the players in the camera's field of view. The mapping between the input and output is computed by a collection of classifiers that operate in real time, and using as little as possible of the Xbox CPU.

One of these classifiers (Shotton et al., 2011) was trained from a massive dataset using supervised learning; the ground truth data used for the learning process is obtained from a motion capture device, similar to the ones used at movie studios for digitizing the movements of actors. The training is essentially performed on millions of pairs of video frames annotated with the correct joint positions.

While the core algorithms are essentially simple, the actual implementation requires substantial tuning to perform efficiently, due to the immense amount of training data. For example, we cannot afford to explicitly materialize all the features used for training; instead the features are represented implicitly, and computed on demand. The data structures manipulated are multi-dimensional and sparse; a substantial amount of code deals with manipulating efficiently the distributed sparse representations; moreover, as the sparsity of the data structures changes dynamically during the training process (some dimensions become progressively denser), the underlying data representation is also changed dynamically.

The implementation of these algorithms has stretched the capabilities of DryadLINQ and uncovered several performance limitations which have in the meantime been fixed. For example, some of the objects represented become very large (several gigabytes/object). There is a tension between obtaining good utilization for all cores (using PLINQ) and having enough RAM to keep all the required state in memory. This required us to change the algorithms performing data buffering and to override PLINQ's partitioning decisions.

To implement this application we have made use of several DryadLINQ features which we have not presented in this document, that allow us to tune the partitioning of the data and to control the granularity of state and the shape of the query plan. We make important use of .Net libraries, e.g., to parse the video/image input format. We have also implemented (with very little effort) workflows of multiple jobs and checkpointing of workflows, which allows us to restart the computation pipelines mid-way.

Overall, DryadLINQ has been an invaluable tool for the Kinect training project; it allowed us to quickly prototype the algorithms and to execute them at scale on an essentially unreliable medium (at this scale failures become frequent enough to make a simple-minded solution completely impractical).

1.4.4 Availability

Dryad, DryadLINQ, and the machine learning code from this chapter is available for download from the DryadLINQ project page: `http://research.microsoft.com/dryadlinq/`. A commercial implementation of DryadLINQ called LINQ to HPC is at the time of the writing available in Beta 2 at `http://msdn.microsoft.com/en-us/library/hh378101.aspx`.

References

Dean, Jeff, and Ghemawat, Sanjay. 2004 (Dec.). MapReduce: Simplified Data Processing on Large Clusters. Pages 137–150 of: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*.

Duffy, Joe. 2007 (January). A query language for data parallel programming. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*.

Google. 2010 (Accessed 27 August). *Protocol Buffers*. `http://code.google.com/apis/protocolbuffers/`.

Isard, Michael, Budiu, Mihai, Yu, Yuan, Birrell, Andrew, and Fetterly, Dennis. 2007 (March). Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. Pages 59–72 of: *Proceedings of European Conference on Computer Systems (EuroSys)*.

Microsoft. 2010 (Accessed 27 August). *The LINQ Project*. `http://msdn.microsoft.com/netframework/future/linq/`.

Shotton, Jamie, Fitzgibbon, Andrew, Cook, Mat, Sharp, Toby, Finocchio, Mark, Moore, Richard, Kipman, Alex, and Blake, Andrew. 2011 (June 21-23). Real-Time Human Pose Recognition in Parts from a Single Depth Image. In: *Computer Vision and Pattern Recognition (CVPR)*.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., and Currey, J. 2008 (December 8-10). 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)*.

Yu, Yuan, Gunda, Pradeep Kumar, and Isard, Michael. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. Pages 247–260 of: *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM.