Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations

Yuan Yu
Microsoft Research
1065 La Avenida Ave.
Mountain View, CA 94043
yuanbyu@microsoft.com

Pradeep Kumar Gunda
Microsoft Research
1065 La Avenida Ave.
Mountain View, CA 94043
pgunda@microsoft.com

Michael Isard
Microsoft Research
1065 La Avenida Ave.
Mountain View, CA 94043
misard@microsoft.com

ABSTRACT

Data-intensive applications are increasingly designed to execute on large computing clusters. Grouped aggregation is a core primitive of many distributed programming models, and it is often the most efficient available mechanism for computations such as matrix multiplication and graph traversal. Such algorithms typically require non-standard aggregations that are more sophisticated than traditional built-in database functions such as `Sum` and `Max`. As a result, the ease of programming user-defined aggregations, and the efficiency of their implementation, is of great current interest.

This paper evaluates the interfaces and implementations for user-defined aggregation in several state of the art distributed computing systems: Hadoop, databases such as Oracle Parallel Server, and DryadLINQ. We show that: the degree of language integration between user-defined functions and the high-level query language has an impact on code legibility and simplicity; the choice of programming interface has a material effect on the performance of computations; some execution plans perform better than others on average; and that in order to get good performance on a variety of workloads a system must be able to select between execution plans depending on the computation. The interface and execution plan described in the MapReduce paper, and implemented by Hadoop, are found to be among the worst-performing choices.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms

Design, Languages, Performance

Keywords

Distributed programming, cloud computing, concurrency

1. INTRODUCTION

Many data-mining computations have as a fundamental subroutine a "GroupBy-Aggregate" operation. This takes a dataset, partitions its records into groups according to some key, then performs an aggregation over each resulting group. GroupBy-Aggregate is useful for summarization, e.g. finding average household income by zip code from a census dataset, but it is also at the heart of the distributed implementation of algorithms such as matrix multiplication [22, 27]. The ability to perform GroupBy-Aggregate at scale is therefore increasingly important, both for traditional data-mining tasks and also for emerging applications such as web-scale machine learning and graph analysis.

This paper analyzes the programming models that are supplied for user-defined aggregation by several state of the art distributed systems, evaluates a variety of optimizations that are suitable for aggregations with differing properties, and investigates the interaction between the two. In particular, we show that the choice of programming interface not only affects the ease of programming complex user-defined aggregations, but can also make a material difference to the performance of some optimizations.

GroupBy-Aggregate has emerged as a canonical execution model in the general-purpose distributed computing literature. Systems like MapReduce [9] and Hadoop [3] allow programmers to decompose an arbitrary computation into a sequence of maps and reductions, which are written in a full-fledged high level programming language (C++ and Java, respectively) using arbitrary complex types. The resulting systems can perform quite general tasks at scale, but offer a low-level programming interface: even common operations such as database Join require a sophisticated understanding of manual optimizations on the part of the programmer. Consequently, layers such as Pig Latin [19] and HIVE [1] have been developed on top of Hadoop, offering a

SQL-like programming interface that simplifies common data-processing tasks. Unfortunately the underlying execution plan must still be converted into a sequence of maps and reductions for Hadoop to execute, precluding many standard parallel database optimizations.

Parallel databases [15] have for some time permitted user-defined selection and aggregation operations that have the same computational expressiveness as MapReduce, although with a slightly different interface. For simple computations the user-defined functions are written using built-in languages that integrate tightly with SQL but have restricted type systems and limited ability to interact with legacy code or libraries. Functions of even moderate complexity, however, must be written using external calls to languages such as C and C++ whose integration with the database type system can be difficult to manage [24].

Dryad [16] and DryadLINQ [26] were designed to address some of the limitations of databases and MapReduce. Dryad is a distributed execution engine that lies between databases and MapReduce: it abandons much of the traditional functionality of a database (transactions, in-place updates, etc.) while providing fault-tolerant execution of complex query plans on large-scale clusters. DryadLINQ is a language layer built on top of Dryad that tightly integrates distributed queries into high level .NET programming languages. It provides a unified data model and programming language that support relational queries with user-defined functions. Dryad and DryadLINQ are an attractive research platform because Dryad supports execution plans that are more complex than those provided by a system such as Hadoop, while the DryadLINQ source is available for modification, unlike that of most parallel databases. This paper explains in detail how distributed aggregation can be treated efficiently by the DryadLINQ optimization phase, and extends the DryadLINQ programming interface as well as the set of optimizations the system may apply.

The contributions of this paper are as follows:

- We compare the programming models for user-defined aggregation in Hadoop, DryadLINQ, and parallel databases, and show the impact of interface-design choices on optimizations.
- We describe and implement a general, rigorous treatment of distributed grouping and aggregation in the DryadLINQ system.
- We use DryadLINQ to evaluate several optimization techniques for distributed aggregation in real applications running on a medium-sized cluster of several hundred computers.

The structure of this paper is as follows. Section 2 explains user-defined aggregation and gives an overview of how a GroupBy-Aggregate computation can be distributed. Section 3 describes the programming interfaces for user-defined aggregation offered by the three systems we consider, and Section 4 outlines the programs we use for our evaluation. Section 5 presents several implementation strategies which are then evaluated in Section 6 using a variety of workloads. Section 7 surveys related work, and Section 8 contains a discussion and conclusions.

2. DISTRIBUTED AGGREGATION

This section discusses the functions that must be supplied in order to perform general-purpose user-defined aggregations. Our example execution plan shows a map followed by an aggregation, however in general an aggregation might, for example, consume the output of more complex processing such as a Join or a previous aggregation. We explain the concepts using the iterator-based programming model adopted by MapReduce [9] and Hadoop [19], and discuss alternatives used by parallel databases and DryadLINQ below in Section 3. We use an integer-average computation as a running example. It is much simpler than most interesting user-defined aggregations, and is included as a primitive in many systems, however its implementation has the same structure as that of many much more complex functions.

2.1 User-defined aggregation

The MapReduce programming model [9] supports grouped aggregation using a user-supplied functional programming primitive called `Reduce`:

- `Reduce`: $\langle K, \textit{Sequence of } R\rangle \rightarrow \textit{Sequence of } S$ takes a sequence of records of type R, all with the same key of type K, and outputs zero or more records of type S.

Here is the pseudocode for a `Reduce` function to compute integer average:

```
double Reduce(Key k, Sequence<int> recordSequence)
{
  // key is ignored
  int count = 0, sum = 0;
  foreach (r in recordSequence) {
    sum += r; ++count;
  }
  return (double)sum / (double)count;
}
```

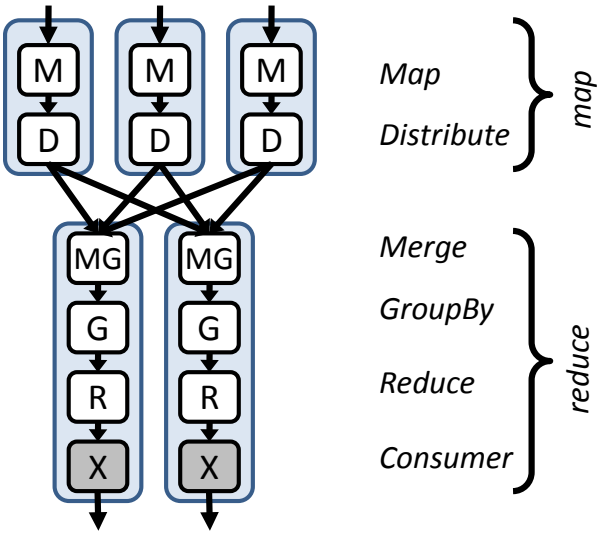

Figure 1: Distributed execution plan for MapReduce when `reduce` cannot be decomposed to perform partial aggregation.

With this user-defined function, and merge and grouping operators provided by the system, it is possible to execute a simple distributed computation as shown in Figure 1. The computation has exactly two phases: the first phase executes a `Map` function on the inputs to extract keys and records, then performs a partitioning of these outputs based on the keys of the records. The second phase collects and merges all the records with the same key, and passes them to the `Reduce` function. (This second phase is equivalent to `GroupBy` followed by `Aggregate` in the database literature.)

As we shall see in the following sections, many optimizations for distributed aggregation rely on computing and combining "partial aggregations." Suppose that aggregating the sequence R_k of all the records with a particular key k results in output S_k. A partial aggregation computed from a subsequence r of R_k is an intermediate result with the property that partial aggregations of all the subsequences of R_k can be combined to generate S_k. Partial aggregations may exist, for example, when the aggregation function is commutative and associative, and Section 2.2 below formalizes the notion of decomposable functions which generalize this case. For our running example of integer average, a partial aggregate contains a partial sum and a partial count:

```
struct Partial {
  int partialSum;
  int partialCount;
}
```

Often the partial aggregation of a subsequence r is much smaller than r itself: in the case of average for example the partial sum is just two values,

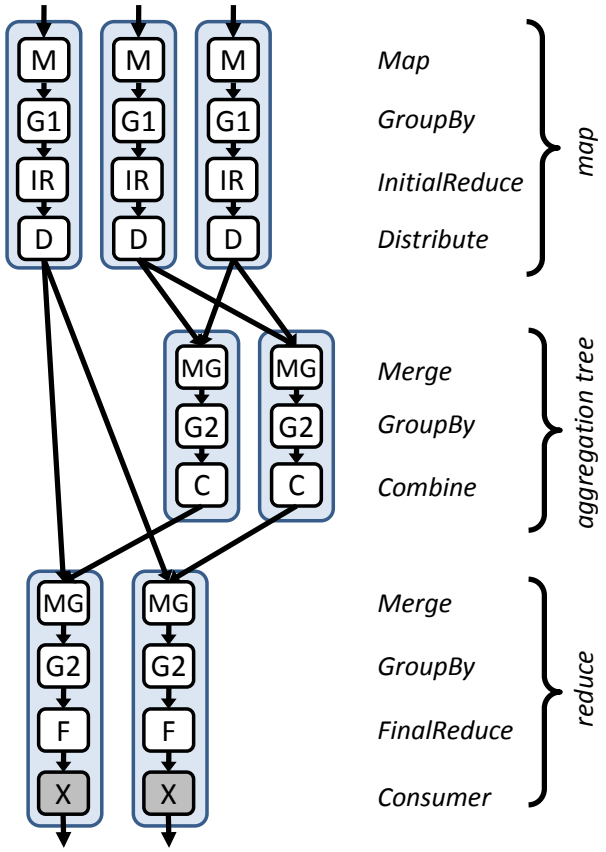

Figure 2: Distributed execution plan for MapReduce when `reduce` supports partial aggregation. The implementation of GroupBy in the first stage may be different to that in the later stages, as discussed in Section 5.

regardless of the number of integers that have been processed. When there is such substantial data reduction, partial aggregation can be introduced both as part of the initial Map phase and in an aggregation tree, as shown in Figure 2, to greatly reduce network traffic. In order to decompose a user-defined aggregation using partial aggregation it is necessary to introduce auxiliary functions, called "Combiners" in [9], that synthesize the intermediate results into the final output. The MapReduce system described in [9] can perform partial aggregation on each local computer before transmitting data across the network, but does not use an aggregation tree.

In order to enable partial aggregation a user of MapReduce must supply three functions:

1. `InitialReduce`: $\langle K, \textit{Sequence of } R\rangle \rightarrow \langle K, X\rangle$ which takes a sequence of records of type R, all with the same key of type K, and outputs a partial aggregation encoded as the key of type K and an intermediate type X.

2. `Combine`: $\langle K, \textit{Sequence of } X\rangle \rightarrow \langle K, X\rangle$ which takes a sequence of partial aggregations of type

X, all with the same key of type K, and outputs a new, combined, partial aggregation once again encoded as an object of type X with the shared key of type K.

3. `FinalReduce`: $\langle K, \textit{Sequence of } X\rangle \rightarrow \textit{Sequence of } S$ which takes a sequence of partial aggregations of type X, all with the same key of type K, and outputs zero or more records of type S.

In simple cases such as `Sum` or `Min` the types R, X and S are all the same, and `InitialReduce`, `Combine` and `FinalReduce` can all be computed using the same function. Three separate functions are needed even for straightforward computations such as integer average:

```
Partial InitialReduce(Key k,
                      Sequence<int> recordSequence) {
  Partial p = { 0, 0 };
  foreach (r in recordSequence) {
    p.partialSum += r;
    ++p.partialCount;
  }
  return <k, p>;
}

Partial Combine(Key k,
                Sequence<Partial> partialSequence) {
  Partial p = { 0, 0 };
  foreach (r in partialSequence) {
    p.partialSum += r.partialSum;
    p.partialCount += r.partialCount;
  }
  return <k, p>;
}

double FinalReduce(Key k,
                   Sequence<Partial> partialSequence)
{ // key is ignored
  Partial p = Combine(k, partialSequence);
  return (double)p.partialSum /
         (double)p.partialCount;
}
```

2.2 Decomposable functions

We can formalize the above discussion by introducing the notion of decomposable functions.

DEFINITION 1. *We use $\overline{x}$ to denote a sequence of data items, and use $\overline{x}_1 \oplus \overline{x}_2$ to denote the concatenation of $\overline{x}_1$ and $\overline{x}_2$. A function H is decomposable if there exist two functions I and C satisfying the following conditions:*

1) *H is the composition of I and C: $\forall \overline{x}_1, \overline{x}_2 : H(\overline{x}_1 \oplus \overline{x}_2) = C(I(\overline{x}_1 \oplus \overline{x}_2)) = C(I(\overline{x}_1) \oplus I(\overline{x}_2))$*

2) *I is commutative: $\forall \overline{x}_1, \overline{x}_2 : I(\overline{x}_1 \oplus \overline{x}_2) = I(\overline{x}_2 \oplus \overline{x}_1)$*

3) *C is commutative: $\forall \overline{x}_1, \overline{x}_2 : C(\overline{x}_1 \oplus \overline{x}_2) = C(\overline{x}_2 \oplus \overline{x}_1)$*

DEFINITION 2. *A function H is associative-decomposable if there exist two functions I and C satisfying conditions 1–3 above, and in addition C is associative: $\forall \overline{x}_1, \overline{x}_2, \overline{x}_3 : C(C(\overline{x}_1 \oplus \overline{x}_2) \oplus \overline{x}_3) = C(\overline{x}_1 \oplus C(\overline{x}_2 \oplus \overline{x}_3))$*

If an aggregation computation can be represented as a set of associative-decomposable functions followed by some final processing, then it can be split up in such a way that the query plan in Figure 2 can be applied. If the computation is instead formed from decomposable functions followed by final processing then the plan from Figure 2 can be applied, but without any intermediate aggregation stages. If the computation is not decomposable then the plan from Figure 1 is required.

Intuitively speaking, I and C correspond to the `InitialReduce` and `Combine` functions for MapReduce that were described in the preceding section. However, there is a small but important difference. Decomposable functions define a class of functions with certain algebraic properties without referring to the aggregation-specific key. This separation of the key type from the aggregation logic makes it possible for the system to automatically optimize the execution of complex reducers that are built up from a combination of decomposable functions, as we show below in Section 3.4.

3. PROGRAMMING MODELS

This section compares the programming models for user-defined aggregation provided by the Hadoop system, a distributed SQL database, and DryadLINQ. We briefly note differences in the way that user-defined aggregation is integrated into the query language in each model, but mostly concentrate on how the user specifies the decomposition of the aggregation computation so that distributed optimizations like those in Figure 2 can be employed. Section 5 discusses how the decomposed aggregation is implemented in a distributed execution.

The systems we consider adopt two different styles of interface for user-defined aggregation. The first is iterator-based, as in the examples in Section 2—the user-defined aggregation function is called once and supplied with an iterator that can be used to access all the records in the sequence. The second is accumulator-based. In this style, which is covered in more detail below in Section 3.2, each partial aggregation is performed by an object that is initialized before first use then repeatedly called with either a singleton record to be accumulated, or another partial-aggregation object to be combined. The iterator-based and accumulator-based interfaces have the same computational expressiveness, how-

ever as we shall see in Section 5 the choice has a material effect on the efficiency of different implementations of `GroupBy`. While there is an automatic and efficient translation from the accumulator interface to the iterator interface, the other direction in general appears to be much more difficult.

3.1 User-defined aggregation in Hadoop

The precise function signatures used for combiners are not stated in the MapReduce paper [9] however they appear to be similar to those provided by the Pig Latin layer of the Hadoop system [19]. The Hadoop implementations of `InitialReduce`, `Combine` and `FinalReduce` for integer averaging are provided in Figure 3. The functions are supplied as overrides of a base class that deals with system-defined "container" objects `DataAtom`, corresponding to an arbitrary record, and `Tuple`, corresponding to a sequence of records. The user is responsible for understanding these types, using casts and accessor functions to fill in the required fields, and manually checking that the casts are valid. This circumvents to some degree the strong static typing of Java and adds substantial apparent complexity to a trivial computation like that in Figure 3, but of course for more interesting aggregation functions the overhead of casting between system types will be less noticeable, and the benefits of having access to a full-featured high-level language, in this case Java, will be more apparent.

3.2 User-defined aggregation in a database

MapReduce can be expressed in a database system that supports user-defined functions and aggregates as follows:

```
SELECT Reduce()
  FROM (SELECT Map() FROM T) R
  GROUPBY R.key
```

where `Map` is a user-defined function outputting to a temporary table R whose rows contain a key `R.key`, and `Reduce` is a user-defined aggregator. (The statement above restricts `Map` and `Reduce` to each produce a single output per input row, however many databases support "table functions" [2, 12] which relax this constraint.) Such user-defined aggregators were introduced in Postgres [23] and are supported in commercial parallel database systems including Oracle and Teradata. Database interfaces for user-defined aggregation are typically object-oriented and accumulator-based, in contrast to the iterator-based Hadoop approach above. For example, in Oracle the user must supply four methods:

1. `Initialize`: This is called once before any data is supplied with a given key, to initialize

```
// InitialReduce: input is a sequence of raw data tuples;
// produces a single intermediate result as output
static public class Initial extends EvalFunc<Tuple> {
  @Override public void exec(Tuple input, Tuple output)
      throws IOException {
    try {
      output.appendField(new DataAtom(sum(input)));
      output.appendField(new DataAtom(count(input)));
    } catch(RuntimeException t) {
      throw new RuntimeException([...]);
    }
  }
}

// Combiner: input is a sequence of intermediate results;
// produces a single (coalesced) intermediate result
static public class Intermed extends EvalFunc<Tuple> {
  @Override public void exec(Tuple input, Tuple output)
      throws IOException {
    combine(input.getBagField(0), output);
  }
}

// FinalReduce: input is one or more intermediate results;
// produces final output of aggregation function
static public class Final extends EvalFunc<DataAtom> {
  @Override public void exec(Tuple input, DataAtom output)
      throws IOException {
    Tuple combined = new Tuple();
    if(input.getField(0) instanceof DataBag) {
      combine(input.getBagField(0), combined);
    } else {
      throw new RuntimeException([...]);
    }
    double sum = combined.getAtomField(0).numval();
    double count = combined.getAtomField(1).numval();
    double avg = 0;
    if (count > 0) {
      avg = sum / count;
    }
    output.setValue(avg);
  }
}

static protected void combine(DataBag values, Tuple output)
      throws IOException {
    double sum = 0;
    double count = 0;
    for (Iterator it = values.iterator(); it.hasNext();) {
      Tuple t = (Tuple) it.next();
      sum += t.getAtomField(0).numval();
      count += t.getAtomField(1).numval();
    }
    output.appendField(new DataAtom(sum));
    output.appendField(new DataAtom(count));
  }

static protected long count(Tuple input)
      throws IOException {
    DataBag values = input.getBagField(0);
    return values.size();
}

static protected double sum(Tuple input)
      throws IOException {
    DataBag values = input.getBagField(0);
    double sum = 0;
    for (Iterator it = values.iterator(); it.hasNext();) {
      Tuple t = (Tuple) it.next();
      sum += t.getAtomField(0).numval();
    }
    return sum;
}
```

Figure 3: **A user-defined aggregator to implement integer averaging in Hadoop.** The supplied functions are conceptually simple, but the user is responsible for marshalling between the underlying data and system types such as `DataAtom` and `Tuple` for which we do not include full definitions here.

```
STATIC FUNCTION ODCIAggregateInitialize
  ( actx IN OUT AvgInterval
  ) RETURN NUMBER IS
  BEGIN
    IF actx IS NULL THEN
      actx := AvgInterval (INTERVAL '0 0:0:0.0' DAY TO
                                   SECOND, 0);
    ELSE
      actx.runningSum := INTERVAL '0 0:0:0.0' DAY TO SECOND;
      actx.runningCount := 0;
    END IF;
    RETURN ODCIConst.Success;
  END;

MEMBER FUNCTION ODCIAggregateIterate
  ( self IN OUT AvgInterval,
    val IN DSINTERVAL_UNCONSTRAINED
  ) RETURN NUMBER IS
  BEGIN
    self.runningSum := self.runningSum + val;
    self.runningCount := self.runningCount + 1;
    RETURN ODCIConst.Success;
  END;

MEMBER FUNCTION ODCIAggregateMerge
  (self IN OUT AvgInterval,
   ctx2 IN AvgInterval
  ) RETURN NUMBER IS
  BEGIN
    self.runningSum := self.runningSum + ctx2.runningSum;
    self.runningCount := self.runningCount +
                              ctx2.runningCount;
    RETURN ODCIConst.Success;
  END;

MEMBER FUNCTION ODCIAggregateTerminate
  ( self IN AvgInterval,
    ReturnValue OUT DSINTERVAL_UNCONSTRAINED,
    flags IN NUMBER
  ) RETURN NUMBER IS
  BEGIN
    IF self.runningCount <> 0 THEN
      returnValue := self.runningSum / self.runningCount;
    ELSE
      returnValue := self.runningSum;
    END IF;
    RETURN ODCIConst.Success;
  END;
```

Figure 4: **A user-defined combiner in the Oracle database system that implements integer averaging.** This example is taken from `http://www.oracle.com/technology/oramag/oracle/06-jul/o46sql.html`.

the state of the aggregation object.

2. `Iterate`: This may be called multiple times, each time with a single record with the matching key. It causes that record to be accumulated by the aggregation object.

3. `Merge`: This may be called multiple times, each time with another aggregation object with the matching key. It combines the two partial aggregations.

4. `Final`: This is called once to output the final record that is the result of the aggregation.

Figure 4 shows an implementation of integer average as an Oracle user-defined aggregator. For functions like average, whose types map well to SQL base types and which can be written entirely using Oracle's built-in extension language, the type-integration is better than that of Hadoop. However if the user-defined functions and types are more complex and must be implemented in a full-fledged language such as C/C++, the database implementation becomes substantially more difficult to understand and manage [24].

3.3 User-defined aggregation in the DryadLINQ system

DryadLINQ integrates relational operators with user code by embedding the operators in an existing language, rather than calling into user-defined functions from within a query language like Pig Latin or SQL. A distributed grouping and aggregation can be expressed in DryadLINQ as follows:

```
var groups = source.GroupBy(KeySelect);
var reduced = groups.SelectMany(Reduce);
```

In this fragment, `source` is a DryadLINQ collection (which is analagous to a SQL table) of .NET objects of type R. `KeySelect` is an expression that computes a key of type K from an object of type R, and `groups` is a collection in which each element is a "group" (an object of type `IGrouping<K,R>`) consisting of a key of type K and a collection of objects of type R. Finally, `Reduce` is an expression that transforms an element of `groups` into a sequence of zero or more objects of type S, and `reduced` is a collection of objects of type S. DryadLINQ programs are statically strongly typed, so the `Reduce` expression could for example be any function that takes an object of type `IGrouping<K,R>` and returns a collection of objects of type S, and no type-casting is necessary. Aggregation without grouping is expressed in DryadLINQ using the `Aggregate` operator. We added a new overloaded `Aggregate` operator to DryadLINQ to mirror the use of `Select` since the standard LINQ `Aggregate` operator uses a slightly different interface.

We have implememented both accumulator- and iterator-based interfaces for user-defined aggregation in DryadLINQ. We first describe the iterator-based interface in some detail, then briefly outline the accumulator based style.

Iterator-based aggregation. We hard-coded into DryadLINQ the fact that standard functions such as `Max` and `Sum` are associative-decomposable and we added the following annotation syntax

```
[AssociativeDecomposable("I", "C")]
public static X H(IEnumerable<R> g) {
  [ ... ]
}
```

```
public static IntPair InitialReduce(IEnumerable<int> g) {
  return new IntPair(g.Sum(), g.Count());
}

public static IntPair Combine(IEnumerable<IntPair> g) {
  return new IntPair(g.Select(x => x.first).Sum(),
                     g.Select(x => x.second).Sum());
}

[AssociativeDecomposable("InitialReduce", "Combine")]
public static IntPair PartialSum(IEnumerable<int> g) {
  return InitialReduce(g);
}

public static double Average(IEnumerable<int> g) {
  IntPair final = g.Aggregate(x => PartialSum(x));
  if (final.second == 0) return 0.0;
  return (double)final.first / (double)final.second;
}
```

Figure 5: **An iterator-based implementation of `Average` in DryadLINQ that uses an associative-decomposable subroutine `PartialSum`.** The annotation on `PartialSum` indicates that the system may split the computation into calls to the two functions `InitialReduce` and `Combine` when executing a distributed expression plan.

which a programmer can use to indicate that a function `H` is associative-decomposable with respect to iterator-based functions `I` and `C`, along with a similar annotation to indicate a `Decomposable` function. The DryadLINQ implementation of iterator-based integer averaging is shown in Figure 5. The implementations match the Hadoop versions in Figure 3 quite closely, but DryadLINQ's tighter language integration means that no marshaling is necessary. Note also the LINQ idiom in `InitialReduce` and `Combine` of using subqueries instead of loops to compute sums and counts.

Accumulator-based aggregation. We also implemented support for an accumulator interface for partial aggregation. The user must define three static functions:

```
public X Initialize();
public X Iterate(X partialObject, R record);
public X Merge(X partialObject, X objectToMerge);
```

where `X` is the type of the object that is used to accumulate the partial aggregation, and supply them using a three-argument variant of the `AssociativeDecomposable` annotation. Figure 6 shows integer averaging using DryadLINQ's accumulator-based interface.

3.4 Aggregating multiple functions

We implemented support within DryadLINQ to automatically generate the equivalent of combiner functions in some cases. We define a reducer in DryadLINQ to be an expression that maps an `IEnumerable` or `IGrouping` object to a sequence of objects of some other type.

```
public static IntPair Initialize() {
  return new IntPair(0, 0);
}

public static IntPair Iterate(IntPair x, int r) {
  x.first += r;
  x.second += 1;
  return x;
}

public static IntPair Merge(IntPair x, IntPair o) {
  x.first += o.first;
  x.second += o.second;
  return x;
}

[AssociativeDecomposable("Initialize", "Iterate", "Merge")]
public static IntPair PartialSum(IEnumerable<int> g) {
  return new IntPair(g.Sum(), g.Count());
}

public static double Average(IEnumerable<int> g) {
  IntPair final = g.Aggregate(x => PartialSum(x));
  if (final.second == 0) return 0.0;
  else return (double)final.first / (double)final.second;
}
```

Figure 6: **An accumulator-based implementation of `Average` in DryadLINQ that uses an associative-decomposable subroutine `PartialSum`.** The annotation on `PartialSum` indicates that the system may split the computation into calls to the three functions `Initialize`, `Iterate` and `Merge` when executing a distributed expression plan.

DEFINITION 3. *Let* `g` *be the formal argument of a reducer. A reducer is decomposable if every terminal node of its expression tree satisfies one of the following conditions:*

1) *It is a constant or, if* `g` *is an* `IGrouping`*, of the form* `g.Key`*, where* `Key` *is the property of the* `IGrouping` *interface that returns the group's key.*

2) *It is of the form* `H(g)` *for a decomposable function* `H`*.*

3) *It is a constructor or method call whose arguments each recursively satisfies one of these conditions.*

Similarly a reducer is associative-decomposable if it can be broken into associative-decomposable functions.

It is a common LINQ idiom to write a statement such as

```
var reduced = groups.
  Select(x => new T(x.Key, x.Sum(), x.Count()));
```

The expression inside the `Select` statement in this example is associative-decomposable since `Sum` and `Count` are system-defined associative-decomposable functions. When DryadLINQ encounters a statement like this we use reflection to discover all the decomposable function calls in the reducer's expression tree and their decompositions. In this example the decomposable functions are `Sum` with decomposition `I=Sum, C=Sum` and `Count` with decomposition `I=Count, C=Sum`.

Our system will automatically generate `InitialReduce`, `Combine` and `FinalReduce` functions from these decompositions, along with a tuple type to store the partial aggregation. For example, the `InitialReduce` function in this example would compute both the `Sum` and the `Count` of its input records and output a pair of integers encoding this partial sum and partial count. The ability to do this automatic inference on function compositions is very useful, since it allows programmers to reason about and annotate their library functions using Definition 1 independent of their usage in distributed aggregation. Any reducer expression that is composed of built-in and user-annotated decomposable functions will enable the optimization of partial aggregation. A similar automatic combination of multiple aggregations could be implemented by the Pig Latin compiler or a database query planner.

Thus the integer average computation could simply be written

```
public static double Average(IEnumerable<int> g)
{
  IntPair final = g.Aggregate(x =>
                  new IntPair(x.Sum(), x.Count()));
  if (final.second == 0) return 0.0;
  else return (double)final.first /
              (double)final.second);
}
```

and the system would automatically synthesize essentially the same code as is written in Figure 5 or Figure 6 depending on whether the optimizer chooses the iterator-based or accumulator-based implementation.

As a more interesting example, the following code computes the standard deviation of a sequence of integers.

```
g.Aggregate(s => Sqrt(s.Sum(x => x*x) -
                      s.Sum()*s.Sum()))
```

Because `Sum` is an associative-decomposable function, the system automatically determines that the expression passed to `Aggregate` is also associative-decomposable. DryadLINQ therefore chooses the execution plan shown in Figure 2, making use of partial aggregation for efficiency.

4. EXAMPLE APPLICATIONS

This section lists the three DryadLINQ example programs that we will evaluate in Section 6. Each example contains at least one distributed aggregation step, and though the programs are quite simple they further illustrate the use of the user-defined aggregation primitives we introduced in Section 3.3. For conciseness, the examples use LINQ's SQL-style syntax instead of the object-oriented syntax adopted in Section 3.1. All of these programs could be implemented in Pig Latin, native Hadoop or SQL, though perhaps less elegantly in some cases.

4.1 Word Statistics

The first program computes statistics about word occurrences in a corpus of documents.

```
var wordStats =
  from doc in docs
  from wc in from word in doc.words
             group word by word into g
             select new WordCount(g.Key, g.Count()))
  group wc.count by wc.word into g
  select ComputeStats(g.Key, g.Count(),
                      g.Max(), g.Sum());
```

The nested query "`from wc ...`" iterates over each document `doc` in the corpus and assembles a document-specific collection of records `wc`, one for each unique word in `doc`, specifying the word and the number of times it appears in `doc`.

The outer query "`group wc.count...`" combines the per-document collections and computes, for each unique word in the corpus, a group containing all of its per-document counts. So for example if the word "confabulate" appears in three documents in the corpus, once in one document and twice in each of the other two documents, then the outer query would include a group with key "confabulate" and counts $\{1, 2, 2\}$.

The output of the full query is a collection of records, one for each unique word in the collection, where each record is generated by calling the user-defined function `ComputeStats`. In the case above, for example, one record will be the result of calling

```
ComputeStats("confabulate", 3, 2, 5).
```

DryadLINQ will use the execution plan given in Figure 2, since `Count`, `Max` and `Sum` are all associative-decomposable functions. The `Map` phase computes the inner query for each document, and the `InitialReduce`, `Combine` and `FinalReduce` stages together aggregate the triple `(g.Count(), g.Max(), g.Sum())` using automatically generated functions as described in Section 3.4.

4.2 Word Top Documents

The second example computes, for each unique word in a corpus, the three documents that have the highest number of occurences of that word.

```
[AssociativeDecomposable("ITop3", "CTop3")]
public static WInfo[] Top3(IEnumerable<WInfo> g)
{
  return g.OrderBy(x => x.count).Take(3).ToArray();
}

public static WInfo[] ITop3(IEnumerable<WInfo> g)
{
```

```
  return g.OrderBy(x => x.count).Take(3).ToArray();
}

public static WInfo[] CTop3(IEnumerable<WInfo[]> g)
{
  return g.SelectMany(x => x).OrderBy(x => x.count).
           Take(3).ToArray();
}

var tops =
  from doc in docs
  from wc in from word in doc.words
             group word by word into g
             select new WInfo(g.Key, g.URL, g.Count())
  group wc by wc.word into g
  select new WordTopDocs(g.Key, Top3(g))
```

The program first computes the per-document count of occurrences of each word using a nested query as in the previous example, though this time we also record the URL of the document associated with each count. Once again the outer query regroups the computed totals according to unique words across the corpus, but now for each unique word `w` we use the function `Top3` to compute the three documents in which `w` occurs most frequently. While `Top3` is associative-decomposable, our implementation cannot infer its decomposition because we do not know simple rules to infer that operator compositions such as `OrderBy.Take` are associative-decomposable. We therefore use an annotation to inform the system that `Top3` is associative-decomposable with respect to `ITop3` and `CTop3`. With this annotation, DryadLINQ can determine that the expression

```
new WordTopDocs(g.Key, Top3(g))
```

is associative-decomposable, so once again the system adopts the execution plan given in Figure 2. While we only show the iterator-based decomposition of `Top3` here, we have also implemented the accumulator-based form and we compare the two in our evaluation in Section 6.

4.3 PageRank

The final example performs an iterative PageRank computation on a web graph. For clarity we present a simplified implementation of PageRank but interested readers can find more highly optimized implementations in [26] and [27].

```
var ranks = pages.Select(p => new Rank(p.name, 1.0));
for (int i = 0; i < interations; i++)
{
    // join pages with ranks, and disperse updates
    var updates =
         from p in pages
         join rank in ranks on p.name equals rank.name
         select p.Distribute(rank);

    // re-accumulate.
    ranks = from list in updates
            from rank in list
            group rank.rank by rank.name into g
            select new Rank(g.Key, g.Sum());
}
```

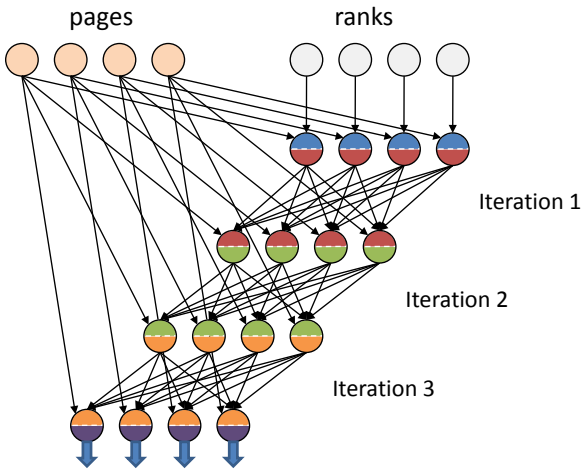

Figure 7: Distributed execution plan for a multi-iteration PageRank computation. Iterations are pipelined together with the final aggregation at the end of one iteration residing in the same process as the Join, rank-distribution, and initial aggregation at the start of the next iteration. The system automatically maintains the partitioning of the rank-estimate dataset and schedules processes to run close to their input data, so the page dataset is never transferred across the network.

Each element `p` of the collection `pages` contains a unique identifier `p.name` and a list of identifiers specifying all the pages in the graph that `p` links to. Elements of `ranks` are pairs specifying the identifier of a page and its current estimated rank. The first statement initializes `ranks` with a default rank for every page in `pages`. Each iteration then calls a method on the page object `p` to distribute `p`'s current rank evenly along its outgoing edges: `Distribute` returns a list of destination page identifiers each with their share of `p`'s rank. Finally the iteration collects these distributed ranks, accumulates the incoming total for each page, and generates a new estimated rank value for that page. One iteration is analogous to a step of MapReduce in which the "Map" is actually a Join pipelined with the distribution of scores, and the "Reduce" is used to re-aggregate the scores. The final select is associative-decomposable so once more DryadLINQ uses the optimized execution plan in Figure 2.

The collection `pages` has been pre-partitioned according to a hash of `p.name`, and the initialization of `ranks` causes that collection to inherit the same partitioning. Figure 7 shows the execution plan for multiple iterations of PageRank. Each iteration computes a new value for `ranks`. Because DryadLINQ knows that `ranks` and `pages` have the same partitioning, the Join in the next iteration can be com-

puted on the partitions of `pages` and `ranks` pairwise without any data re-partitioning. A well-designed parallel database would also be able to automatically select a plan that avoids re-partitioning the datasets across iterations. However, because MapReduce does not natively support multi-input operators such as Join, it is unable to perform a pipelined iterative computation such as PageRank that preserves data locality, leading to much larger data transfer volumes for this type of computation when executed on a system such as Hadoop.

5. SYSTEM IMPLEMENTATION

We now turn our attention to the implementations of distributed reduction for the class of combiner-enabled computations. This section describes the execution plan and six different reduction strategies we have implemented using the DryadLINQ system. Section 6 evaluates these implementations on the applications presented in Section 4.

All our example programs use the execution plan in Figure 2 for their distributed GroupBy-Aggregate computations. This plan contains two aggregation steps: `G1+IR` and `G2+C`. Their implementation has a direct impact on the amount of data reduction at the first stage and also on the degree of pipelining with the preceding and following computations. Our goal of course is to optimize the entire computation, not a single aggregation in isolation. In this section, we examine the implementation choices and their tradeoffs.

We consider the following six implementations of the two aggregation steps, listing them according to the implementation of the first GroupBy (`G1`). All the implementations are multi-threaded to take advantage of our multi-core cluster computers.

FullSort This implementation uses the iterator interface that is described in Section 2.1. The first GroupBy (`G1`) accumulates all the objects in memory and performs a parallel sort on them according to the grouping key. The system then streams over the sorted objects calling `InitialReduce` once for each unique key. The output of the InitialReduce stage remains sorted by the grouping key so we use a parallel merge sort for the Merge operations (`MG`) in the subsequent stages and thus the later GroupBys (`G2`) are simple streaming operations since the records arrive sorted into groups and ready to pass to the Combiners. Since the first stage reads all of the input records before doing any aggregation it attains an optimal data reduction for each partition. However the fact that it accumulates every record in memory before sorting completes makes the strategy unsuitable if the output of the upstream computation is large. Since `G2` is stateless it can be pipelined with a downstream computation as long as `FinalReduce` does not use a large amount of memory. Either the accumulator- or iterator-based interface can be used with this strategy, and we use the iterator-based interface in our experiments. FullSort is the strategy adopted by MapReduce [9] and Hadoop [3].

PartialSort We again use the iterator interface for PartialSort. This scheme reads a bounded number of chunks of input records into memory, with each chunk occupying bounded storage. Each chunk is processed independently in parallel: the chunk is sorted; its sorted groups are passed to `InitialReduce`; the output is emitted; and the next chunk is read in. Since the output of the first stage is not sorted we use non-deterministic merge for `MG`, and we use FullSort for `G2` since we must aggregate all the records for a particular key before calling FinalReduce. PartialSort uses bounded storage in the first stage so it can be pipelined with upstream computations. `G2` can consume unbounded storage, but we expect a large degree of data reduction from pre-aggregation most of the time. We therefore enable the pipelining of downstream computations by default when using PartialSort (and all the following strategies), and allow the user to manually disable it. Since `InitialReduce` is applied independently to each chunk, PartialSort does not in general achieve as much data reduction at the first stage as FullSort. The aggregation tree stage in Figure 2 may therefore be a useful optimization to perform additional data reduction inside a rack before the data are sent over the cluster's core switch.

Accumulator-FullHash This implementation uses the accumulator interface that is described in Section 3.2. It builds a parallel hash table containing one accumulator object for each unique key in the input dataset. When a new unique key is encountered a new accumulator object is created by calling `Initialize`, and placed in the hash table. As each record is read from the input it is passed to the `Iterate` method of its corresponding accumulator object and then discarded. This method makes use of a non-deterministic merge for `MG` and Accumulator-FullHash for `G2`. Storage is proportional to the number of unique keys rather than the num-

ber of records, so this scheme is suitable for some problems for which FullSort would exhaust memory. It is also more general than either sorting method since it only requires equality comparison for keys (as well as the ability to compute an efficient hash of each key). Like FullSort, this scheme achieves optimal data reduction after the first stage of computation. While the iterator-based interface could in principle be used with this strategy it would frequently be inefficient since it necessitates constructing a singleton iterator to "wrap" each input record, creating a new partial aggregate object for that record, then merging it with the partial aggregate object stored in the hash table. We therefore use the accumulator interface in our experiments. Accumulator-FullHash is listed as a GroupBy implementation by the documentation of commercial databases such as IBM DB2 and recent versions of Oracle.

Accumulator-PartialHash This is a similar implementation to Accumulator-FullHash except that it evicts the accumulator object from the hash table and emits its partial aggregation whenever there is a hash collision. Storage usage is therefore bounded by the size of the hash table, however data reduction at the first stage could be very poor for adversarial inputs. We use Accumulator-FullHash for `G2` since we must aggregate all the records for a particular key before calling `FinalReduce`.

Iterator-FullHash This implementation is similar to FullSort in that it accumulates all the records in memory before performing any aggregation, but instead of accumulating the records into an array and then sorting them, Iterator-FullHash accumulates the records into a hash table according to their GroupBy keys. Once all the records have been assembled, each group in the hash table in turn is aggregated and emitted using a single call to `InitialReduce`. `G1` has similar memory characteristics to FullSort, however `G2` must also use Iterator-FullHash because the outputs are not partially sorted. Iterator-FullHash, like Accumulator-FullHash, requires only equality comparison for the GroupBy key.

Iterator-PartialHash This implementation is similar to Iterator-FullHash but, like Accumulator-PartialHash, it emits the group accumulated in the hash table whenever there is a hash collision. It uses bounded storage in the first stage but falls back to Iterator-FullHash for `G2`. Like Accumulator-PartialHash, Iterator-FullHash may result in poor data reduction in its first stage.

In all the implementations, the aggregation tree allows data aggregation according to data locality at multiple levels (computer, rack, and cluster) in the cluster network. Since the aggregation tree is highly dependent on the dynamic scheduling decisions of the vertex processes, it is automatically inserted into the execution graph at run time. This is implemented using the Dryad callback mechanism that allows higher level layers such as DryadLINQ to implement runtime optimization policies by dynamically mutating the execution graph. For the aggregation tree, DryadLINQ supplies the aggregation vertex and policies, and Dryad automatically introduces an aggregation tree based on run time information. Aggregation trees can be particularly useful for PartialSort and PartialHash when the data reduction in the first stage is poor. They are also very beneficial if the input dataset is composed of a lot of small partitions.

Note that while the use of merge sort for `MG` allows FullSort to perform a stateless GroupBy at `G2`, it has a subtle drawback compared to a non-deterministic merge. Merge sort must open all of its inputs at once and interleave reads from them, while non-deterministic merge can read sequentially from one input at a time. This can have a noticeable impact on disk IO performance when there is a large number of inputs.

Although FullSort is the strategy used by MapReduce and Hadoop, the comparison is a little misleading. The Map stage in these systems always operates on one single small input partition at a time, read directly from a distributed file system. It is never pipelined with an upstream computation such as a Join with a large data magnification, or run on a large data partition like the ones in our experiments in the following section. In some ways therefore, MapReduce's FullSort is more like our PartialSort with only computer-level aggregation since it arranges to only read its input in fixed-size chunks. In the evaluation section, we simulated MapReduce in DryadLINQ and compared its performance with our implementations.

As far as we know, neither Iterator-PartialHash nor Accumulator-PartialHash has previously been reported in the literature. However, it should be apparent that there are many more variants on these implementations that could be explored. We have selected this set to represent both established methods and those methods which we have found to perform well in our experiments.

6. EXPERIMENTAL EVALUATION

This section evaluates our implementations of distributed aggregation, focusing on the effectiveness of the various optimization strategies. As explained in Section 4 all of our example programs can be executed using the plan shown in Figure 2. In this plan the stage marked "aggregation tree" is optional and we run experiments with and without this stage enabled. When the aggregation tree is enabled the system performs a partial aggregation within each rack. For larger clusters this single level of aggregation might be replaced by a tree. As noted below, our network is quite well provisioned and so we do not see much benefit from the aggregation tree. In fact it can harm performance, despite the additional data reduction, due to the overhead of starting extra processes and performing additional disk IO. However we also have experience running similar applications on large production clusters with smaller cross-cluster bandwidth, and we have found that in some cases aggregation trees can be essential to get good performance.

We report data reduction numbers for our experiments. In each case a value is reported for each stage of the computation and it is computed as the ratio between the uncompressed size of the total data input to the stage and the uncompressed size of its total output. We report these values to show how much opportunity for early aggregation is missed by our bounded-size strategies compared to the optimal FullSort and FullHash techniques. Our implementation in fact compresses intermediate data, so the data transferred between stages is approximately a factor of three smaller than is suggested by these numbers, which further reduces the benefit of using an aggregation tree.

6.1 Dryad and DryadLINQ

DryadLINQ [26] translates LINQ programs written using .NET languages into distributed computations that can be run on the Dryad cluster-computing system [16]. A Dryad job is a directed acyclic graph where each vertex is a program and edges represent data channels. At run time, vertices are processes communicating with each other through the channels, and each channel is used to transport a finite sequence of data records. Dryad's main job is to efficiently schedule vertex processes on cluster computers and to provide fault-tolerance by re-executing failed or slow processes. The vertex programs, data model, and channel data serialization code are all supplied by higher-level software layers, in this case DryadLINQ. In all our examples vertex processes write their output channel data to local disk storage, and read input channel data from the files written by upstream vertices.

At the heart of the DryadLINQ system is the parallel compiler that generates the distributed execution plan for Dryad to run. DryadLINQ first turns a raw LINQ expression into an execution plan graph (EPG), and goes through several phases of semantics-preserving graph rewriting to optimize the execution plan. The EPG is a "skeleton" of the Dryad data-flow graph that will be executed, and each EPG node is expanded at run time into a set of Dryad vertices running the same computation on different partitions of a dataset. The optimizer uses many traditional database optimization techniques, both static and dynamic. More details of Dryad and DryadLINQ can be found in [16, 17, 26].

6.2 Hardware Configuration

The experiments described in this paper were run on a cluster of 236 computers. Each of these computers was running the Windows Server 2003 64-bit operating system. The computers' principal components were two dual-core AMD Opteron 2218 HE CPUs with a clock speed of 2.6 GHz, 16 GBytes of DDR2 RAM, and four 750 GByte SATA hard drives. The computers had two partitions on each disk. The first, small, partition was occupied by the operating system on one disk and left empty on the remaining disks. The remaining partitions on each drive were striped together to form a large data volume spanning all four disks. The computers were each connected to a Linksys SRW2048 48-port full-crossbar GBit Ethernet local switch via GBit Ethernet. There were between 29 and 31 computers connected to each local switch. Each local switch was in turn connected to a central Linksys SRW2048 switch, via 6 ports aggregated using 802.3ad link aggregation. This gave each local switch up to 6 GBits per second of full duplex connectivity. Our research cluster has fairly high cross-cluster bandwidth, however hierarchical networks of this type do not scale easily since the central switch rapidly becomes a bottleneck. Many clusters are therefore less well provisioned than ours for communication between computers in different racks.

6.3 Word Statistics

In this experiment we evaluate the word statistics application described in Section 4.1 using a collection of 140 million web documents with a total size of 1 TB. The dataset was randomly partitioned into 236 partitions each around 4.2 GB in size, and each cluster computer stored one partition. Each partition contains around 500 million words of which

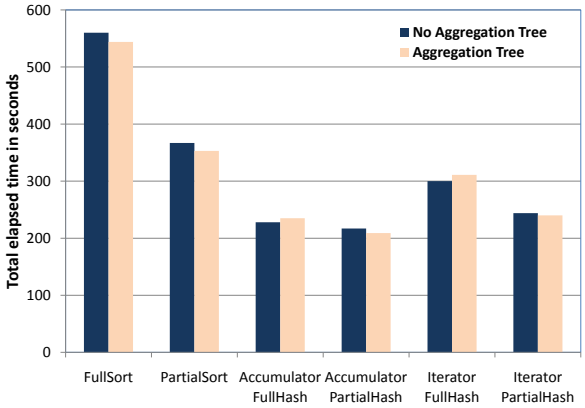

Figure 8: **Time in seconds to compute word statistics with different optimization strategies.**

Reduction strategy	No Aggregation	Aggregation
FullSort	[11.7, 4.5]	[11.7, 2.5, 1.8]
PartialSort	[3.7, 13.7]	[3.7, 7.3, 1.8]
Acc-FullHash	[11.7, 4.5]	[11.7, 2.5, 1.8]
Acc-PartialHash	[4.6, 11.4]	[4.6, 6.15, 1.85]
Iter-FullHash	[11.7, 4.5]	[11.7, 2.5, 1.8]
Iter-PartialHash	[4.1, 12.8]	[4.1, 6.6, 1.9]

Table 1: **Data reduction ratios for the word statistics application under different optimization strategies.**

about 9 million are distinct. We ran this application using the six optimization strategies described in Section 5.

Figure 8 shows the elapsed times in seconds of the six different optimization strategies with and without the aggregation tree. On repeated runs the times were consistent to within 2% of their averages. For all the runs, the majority (around 80%) of the total execution time is spent in the map stage of the computation. The hash-based implementations significantly outperform the others, and the partial reduction implementations are somewhat better than their full reduction counterparts.

Table 1 shows the amount of data reduction at each stage of the six strategies. The first column shows the experimental results when the aggregation tree is turned off. The two numbers in each entry represent the data reductions of the map and reduce stages. The second column shows the results obtained using an aggregation tree. The three numbers in each entry represent the data reductions of the map, aggregation tree, and reduce stages. The total data reduction for a computation is the product of the numbers in its entry. As expected, using PartialHash or PartialSort for the map stage always results in less data reduction for that stage than is attained by their FullHash and FullSort variants. However, their reduced memory footprint (especially for the hash-based approaches whose storage is proportional to the number of records rather than the number of groups) leads to faster processing time and compensates for the inferior data reduction since our network is fast.

We compared the performance for this application with a baseline experiment that uses the execution plan in Figure 1, i.e. with no partial aggregation. We compare against FullSort, so we use FullSort as the GroupBy implementation in the reduce stage. We used the same 236 partition dataset for this experiment, but there is a data magnification in the output of the map stage so we used 472 reducers to prevent FullSort from running out of memory. The map stage applies the map function and performs a hash partition. The reduce stage sorts the data and performs the Groupby-Aggregate computation. The total elapsed execution time is 15 minutes. This is a 346 second (60%) increase in execution time compared to FullSort, which can be explained by the overhead of additional disk and network IO, validating our premise that performing local aggregation can significantly improve the performance of large-scale distributed aggregation. We performed a similar experiment using the plan in Figure 1 with FullHash in the reduce stage, and obtained a similar performance degradation compared to using FullHash with partial aggregation.

6.4 Word Popularity

In this experiment we evaluate the word popularity application described in Section 4.2 using the same 1 TB dataset of web documents as in the previous experiment. We again compared six optimization strategies, with and without the aggregation tree.

Figure 9 and Table 2 show the total elapsed times and data reductions for each strategy. FullSort and Iterator-FullHash could not complete because they ran out of memory. While the input corpus was the same as for the experiment in Section 6.3, this application retains the URL of each document along with its count and this substantially increases the required storage. Accumulator-FullHash was able to complete because it only stores the partially aggregated values of the groups, not the groups themselves. Once again, the aggregation tree achieved a considerable reduction in the data that had to be transmitted between racks to execute the final stage, but gained little in terms of overall performance. The accumulator-based interfaces performed better than the iterator-based interfaces, and Accumulator-PartialHash ran a little faster than Accumulator-FullHash.

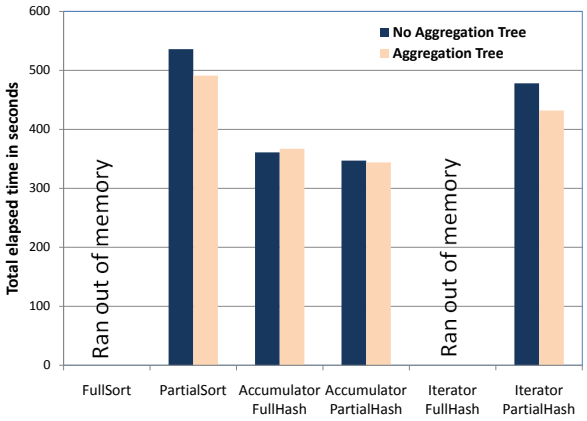

Figure 9: Time in seconds to compute word popularity with different optimization strategies.

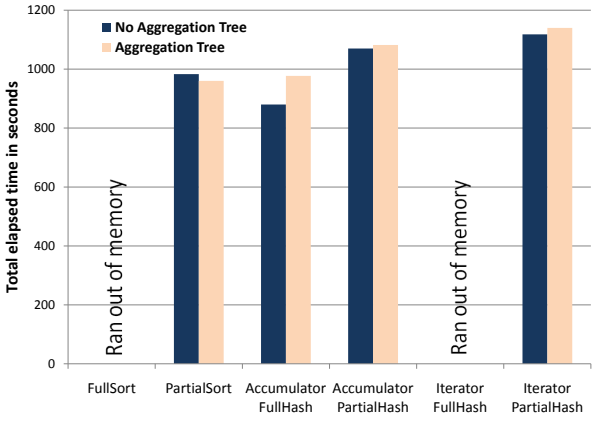

Figure 10: Time in seconds to compute PageRank for one iteration with the six optimization strategies.

Reduction strategy	No Aggregation	Aggregation
FullSort	NC	NC
PartialSort	[3.6, 17.3]	[3.6, 7.3, 2.3]
Acc-FullHash	[11.3, 5.5]	[11.3, 2.5, 2.2]
Acc-PartialHash	[4.4, 14]	[4.4, 6, 2.3]
Iter-FullHash	NC	NC
Iter-PartialHash	[3.8, 16.4]	[3.8, 7.7, 2.12]

Table 2: Data reduction ratios for the word popularity application under different optimization strategies. NC indicates that a result was not computed because the implementation ran out of memory.

6.5 PageRank

In this experiment we evaluate the PageRank computation described in Section 4.3 using a moderate sized web graph. The dataset consists of about 940 million web pages and occupies around 700 GB of storage. For this experiment the dataset was hash partitioned by URL into 472 partitions of around 1.35 GB each, with each cluster computer storing two partitions.

Figure 10 shows the elapsed times in seconds for running a single iteration of PageRank using our six optimization strategies. On repeated runs the times were consistent to within 5% of their averages. This application demonstrates a scenario where a join and a distributed reduction are pipelined together to avoid writing the output of the Join to intermediate storage. The number of records output by the Join is proportional to the number of edges in the graph, and is too large to fit in memory so neither FullSort nor Iterator-FullHash can complete. However the number of groups is only proportional to the number of pages, so Accumulator-FullHash succeeds. Table 3 shows the data reduction of the various stages of the computation which is lower than that of the previous two examples since the average number of elements in each group is smaller.

Reduction strategy	No Aggregation	Aggregation
FullSort	NC	NC
PartialSort	[1.28, 4.1]	[1.28, 2.85, 1.4]
Acc-FullHash	[2.2, 2.4]	[2.2, 1.8, 1.3]
Acc-PartialHash	[1.7, 3.1]	[1.7, 2.4, 1.3]
Iter-FullHash	NC	NC
Iter-PartialHash	[1.75, 3]	[1.75, 2.4, 1.25]

Table 3: Data reductions of pagerank with the six optimization strategies.

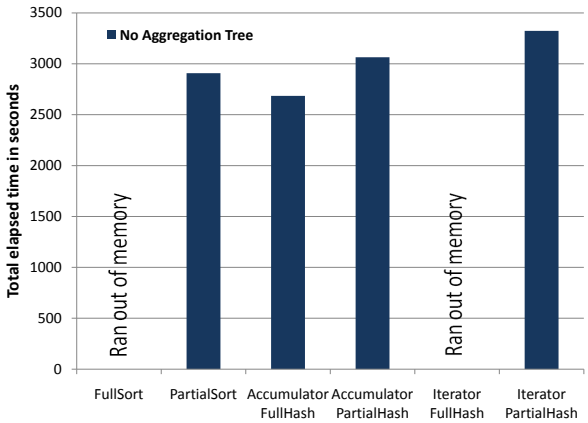

Figure 11: Time in seconds to compute PageRank for three iterations with the six optimization strategies.

Figure 11 shows the elapsed times in seconds for running an application that performs three iterations of the PageRank computation. We only report results with the aggregation tree disabled since it was shown not to be beneficial in the one-iteration case. In all cases the total running time is slightly less than three times that of the corresponding single-iteration experiment.

6.6 Comparison with MapReduce

This section reports our performance comparison with MapReduce. We simulated two possible im-

plementations of MapReduce (denoted MapReduce-I and MapReduce-II) in DryadLINQ. The implementations differ only in their map phase. MapReduce-I applies the Map function, sorts the resulting records, and writes them to local disk, while MapReduce-II performs partial aggregation on the sorted records before outputting them. Both implementations perform computer-level aggregation after the map stage, and the reduce stage simply performs a merge sort and applies the reduce function. We evaluated the two implementations on the word statistics application from Section 6.3, where the input dataset was randomly partitioned into 16000 partitions each approximately 64 MB in size. Each implementation executed 16000 mapper processes and 236 reducer processes.

The two MapReduce implementations have almost identical performance on our example, each taking just over 700 seconds. Comparing to Figure 8, it is clear that they were outperformed by all six implementations described in Section 5. The MapReduce implementations took about three times longer than the best strategy (Accumulator-PartialHash), and twice as long as PartialSort which is the most similar to MapReduce as noted in Section 5. The bulk of the performance difference is due to the overhead of running tens of thousands of short-lived processes.

6.7 Analysis

In all experiments the accumulator-based interfaces perform best, which may explain why this style of interface was chosen by the database community. The implementations that use bounded memory at the first stage, but achieve lower data reduction, complete faster in our experiments than those which use more memory, but output less data, in the first stage. As discussed above, this and the fact that the aggregation tree is not generally effective may be a consequence of our well-provisioned network, and for some clusters performing aggressive early aggregation might be more effective.

Based on these experiments, if we had to choose a single implementation strategy it would be Accumulator-FullHash, since it is faster than the alternatives for PageRank, competitive for the other experiments, and achieves a better early data reduction than Accumulator-PartialHash. However since it does not use bounded storage there are workloads (and computer configurations) for which it cannot be used, so a robust system must include other strategies to fall back on.

The MapReduce strategy of using a very large number of small input partitions performs substantially worse than the other implementations we tried due to the overhead of starting a short-lived process for each of the partitions.

7. RELATED WORK

There is a large body of work studying aggregation in the parallel and distributed computing literature. Our work builds on data reduction techniques employed in parallel databases, cluster data-parallel computing, and functional and declarative programming. To our knowledge, this paper represents the first systematic evaluation of the programming interfaces and implementations of large scale distributed aggregation.

7.1 Parallel and Distributed Databases

Aggregation is an important aspect of database query optimization [6, 14]. Parallel databases [11] such as DB2 [4], Gamma [10], Volcano [13], and Oracle [8] all support pre-aggregation techniques for SQL base types and built-in aggregators. Some systems such as Oracle also support pre-aggregation for user-defined functions. However, when the aggregation involves more complex user-defined functions and data types, the database programming interface can become substantially more difficult to use than DryadLINQ. Databases generally adopt accumulator-based interfaces. As shown in our evaluation, these consistently outperform the iterator interfaces used by systems like MapReduce.

7.2 Cluster Data-Parallel Computing

Infrastructures for large scale distributed data processing have proliferated recently with the introduction of systems such as MapReduce [9], Dryad [16] and Hadoop [3]. All of these systems implement user-defined distributed aggregation, however their interfaces for implementing pre-aggregation are either less flexible or more low-level than that provided by DryadLINQ. No previously published work has offered a detailed description and evaluation of their interfaces and implementations for this important optimization. The work reported in [18] formalizes MapReduce in the context of the Haskell functional programming language.

7.3 Functional and Declarative Languages for Parallel Programming

Our work is also closely related to data aggregation techniques used in functional and declarative parallel programming languages [7, 21, 25]. The formalism of algorithmic skeletons underpins our treatment of decomposable functions in Sections 2 and 3.

The growing importance of data-intensive computation at scale has seen the introduction of a number

of distributed and declarative scripting languages, such as Sawzall [20], SCOPE [5], Pig Latin [19] and HIVE [1]. Sawzall supports user-defined aggregation using MapReduce's combiner optimization. SCOPE supports pre-aggregation for a number of built-in aggregators. Pig Latin supports partial aggregation for algebraic functions, however as explained in Section 3, we believe that the programming interface offered by DryadLINQ is cleaner and easier to use than Pig Latin.

8. DISCUSSION AND CONCLUSIONS

The programming models for MapReduce and parallel databases have roughly equivalent expressiveness for a single MapReduce step. When a user-defined function is easily expressed using a built-in database language the SQL interface is slightly simpler, however more complex user-defined functions are easier to implement using MapReduce or Hadoop. When sophisticated relational queries are required native MapReduce becomes difficult to program. The Pig Latin language simplifies the programming model for complex queries, but the underlying Hadoop platform cannot always execute those queries efficiently. In some ways, DryadLINQ seems to offer the best of the two alternative approaches: a wide range of optimizations; simplicity for common data-processing operations; and generality when computations do not fit into simple types or processing patterns.

Our formulation of partial aggregation in terms of decomposable functions enables us to study complex reducers that are expressed as a combination of simpler functions. However, as noted in Section 4.2, the current DryadLINQ system is not sophisticated enough even to reason about simple operator compositions such as `OrderBy.Take`. We plan to add an analysis engine to the system that will be able to infer the algebraic properties of common operator compositions. This automatic inference should further improve the usability of partial aggregation.

We show that an accumulator-based interface for user-defined aggregation can perform substantially better than an iterator-based alternative. Some programmers may, however, consider the iterator interface to be more elegant or simpler, so may prefer it even though it makes their jobs run slower. Many .NET library functions are also defined in the iterator style. Now that we have implemented both within DryadLINQ we are curious to discover which will be more popular among users of the system.

Another clear finding is that systems should select between a variety of optimization schemes when picking the execution plan for a particular computation, since different schemes are suited to different applications and cluster configurations. Of the three systems we consider, currently only parallel databases are able to do this. Pig Latin and DryadLINQ could both be extended to collect statistics about previous runs of a job, or even to monitor the job as it executes. These statistics could be used as profile-guided costs that would allow the systems' expression optimizers to select between aggregation implementations, and our experiments suggest this would bring substantial benefits for some workloads.

Finally, we conclude that it is not sufficient to consider the programming model or the execution engine of a distributed platform in isolation: it is the system that combines the two that determines how well ease of use can be traded off against performance.

9. ACKNOWLEDGMENTS

We would like to thank the member of the DryadLINQ project for their contributions. We would also like to thank Frank McSherry and Dennis Fetterly for sharing and explaining their implementations of PageRank, and Martín Abadi and Doug Terry for many helpful comments. Thanks also to the SOSP review committee and our shepherd Jon Crowcroft for their very useful feedback.

10. REFERENCES

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

[2] Database Languages—SQL, ISO/IEC 9075-*:2003, 2003.

[3] Hadoop wiki. http://wiki.apache.org/hadoop/, April 2008.

[4] C. Baru and G. Fecteau. An overview of DB2 parallel edition. In *International Conference on Management of Data (SIGMOD)*, pages 460–462, New York, NY, USA, 1995. ACM Press.

[5] 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.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, 1998.

[7] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT

Press, Cambridge, MA, USA, 1991.

[8] T. Cruanes, B. Dageville, and B. Ghosh. Parallel SQL execution in Oracle 10g. In *ACM SIGMOD*, pages 850–854, Paris, France, 2004. ACM.

[9] 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)*, pages 137–150, Dec. 2004.

[10] D. DeWitt, S. Ghandeharizadeh, D. Schneider, H. Hsiao, A. Bricker, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[11] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.

[12] A. Eisenberg, J. Melton, K. Kulkarni, J.-E. Michels, and F. Zemke. Sql:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[13] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD International Conference on Management of data*, pages 102–111, New York, NY, USA, 1990. ACM Press.

[14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[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)*, pages 59–72, March 2007.

[17] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *International Conference on Management of Data (SIGMOD)*, June 29-July 2 2009.

[18] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

[19] 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.

[20] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[21] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[23] L. A. Rowe and M. R. Stonebraker. The postgres data model. In *International Conference of Very Large Data Bases (VLDB)*, pages 83–96. Society Press, 1987.

[24] J. Russell. *Oracle9i Application Developer's Guide—Fundamentals*. Oracle Corporation, 2002.

[25] P. Trinder, H.-W. Loidl, and R. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12((4&5)):469–510, 2002.

[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)*, December 8-10 2008.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, J. Currey, F. McSherry, and K. Achan. Some sample programs written in DryadLINQ. Technical Report MSR-TR-2008-74, Microsoft Research, May 2008.