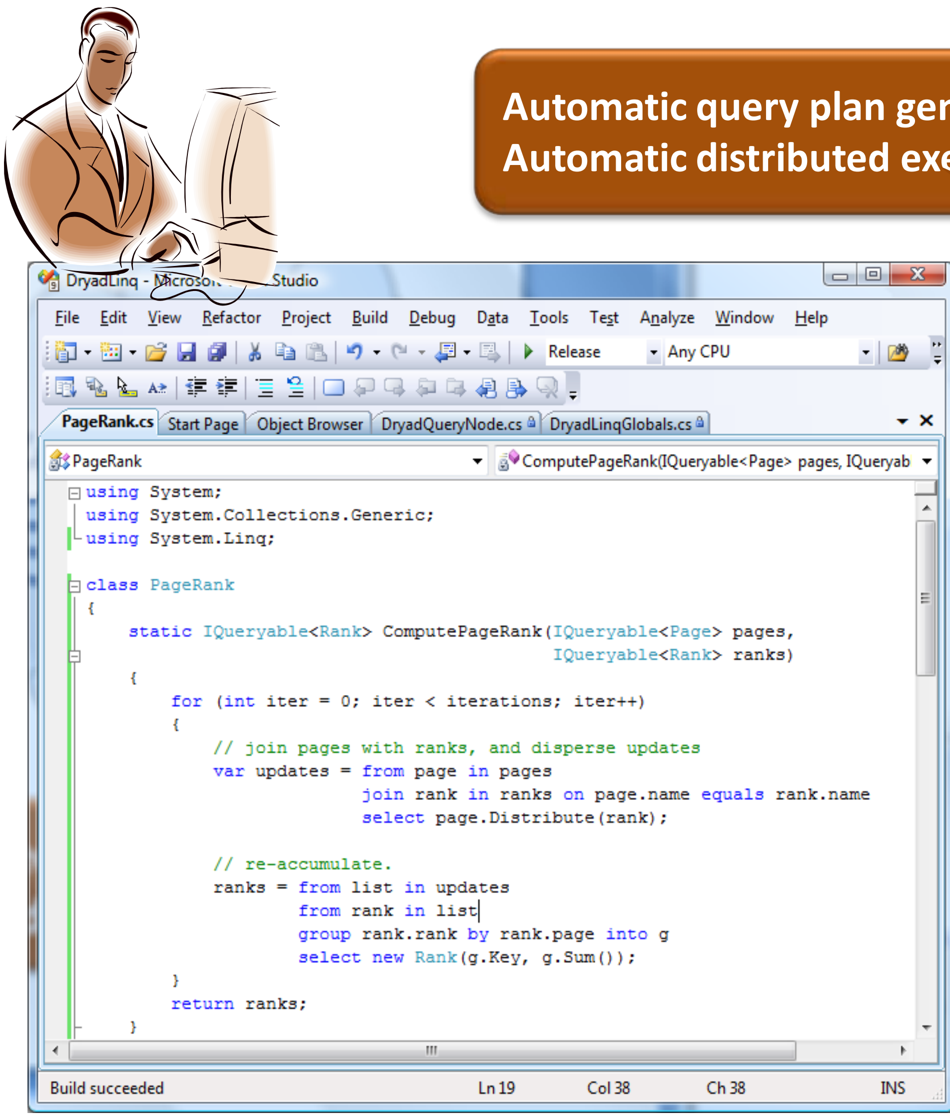
Distributed Aggregation for Data-Parallel Computing Interfaces and Implementations

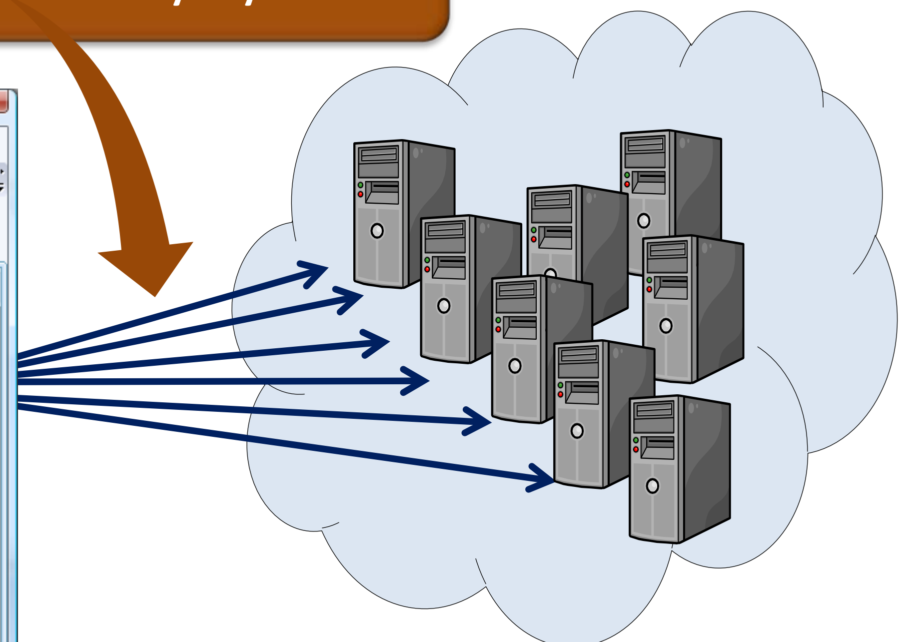
Yuan Yu
Pradeep Kumar Gunda
Michael Isard

Microsoft Research Silicon Valley

Dryad and DryadLINQ

Automatic query plan generation by DryadLINQ
Automatic distributed execution by Dryad





Distributed GroupBy-Aggregate

A core primitive in data-parallel computing

```
source = [upstream computation];
groups = source.GroupBy(keySelector);
reduce = groups.SelectMany(reducer);
result = [downstream computation];
```

Where the programmer defines:

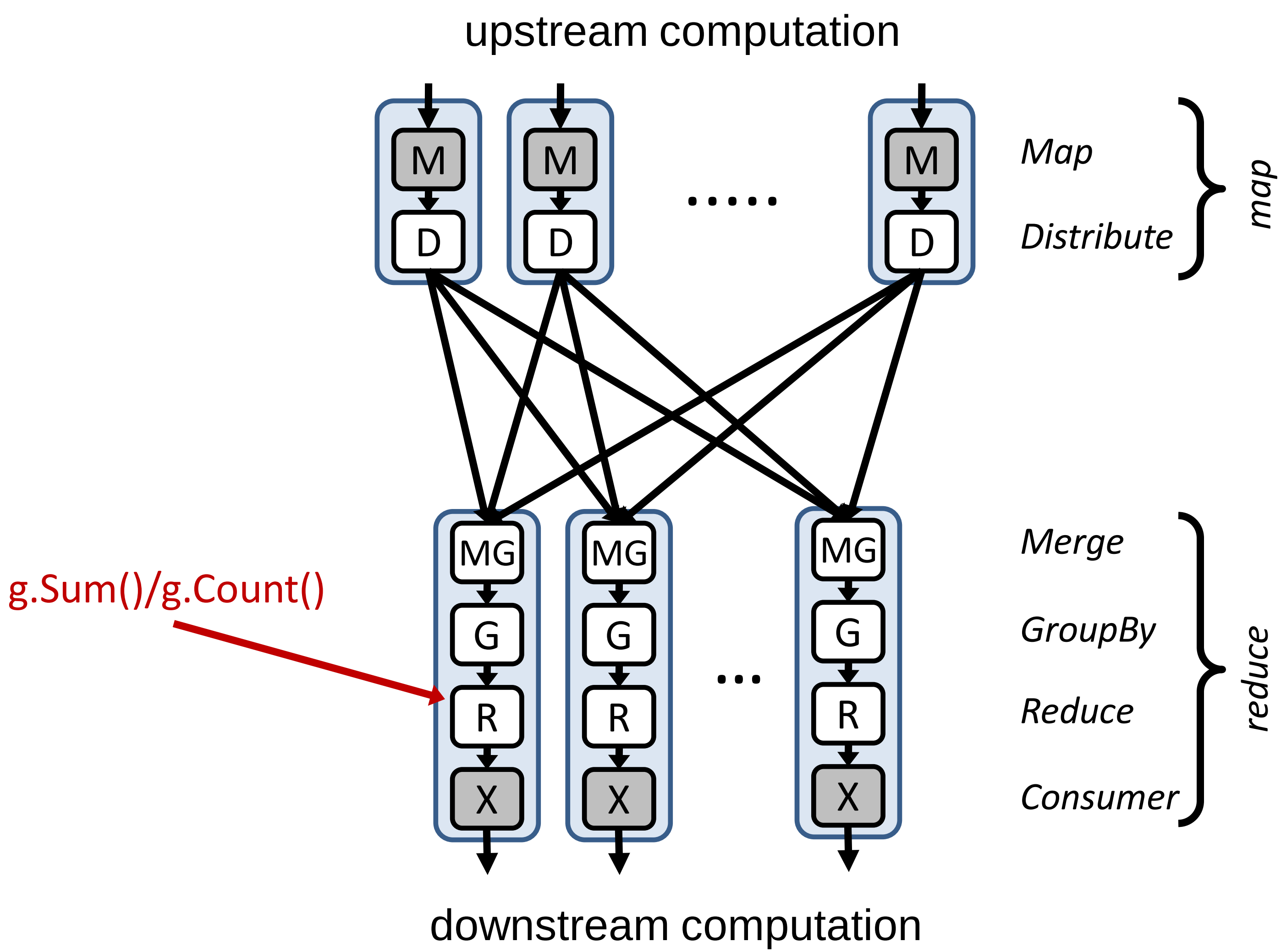
keySelector: T → K
reducer: [K, Seq(T)] → Seq(S)

A Simple Example

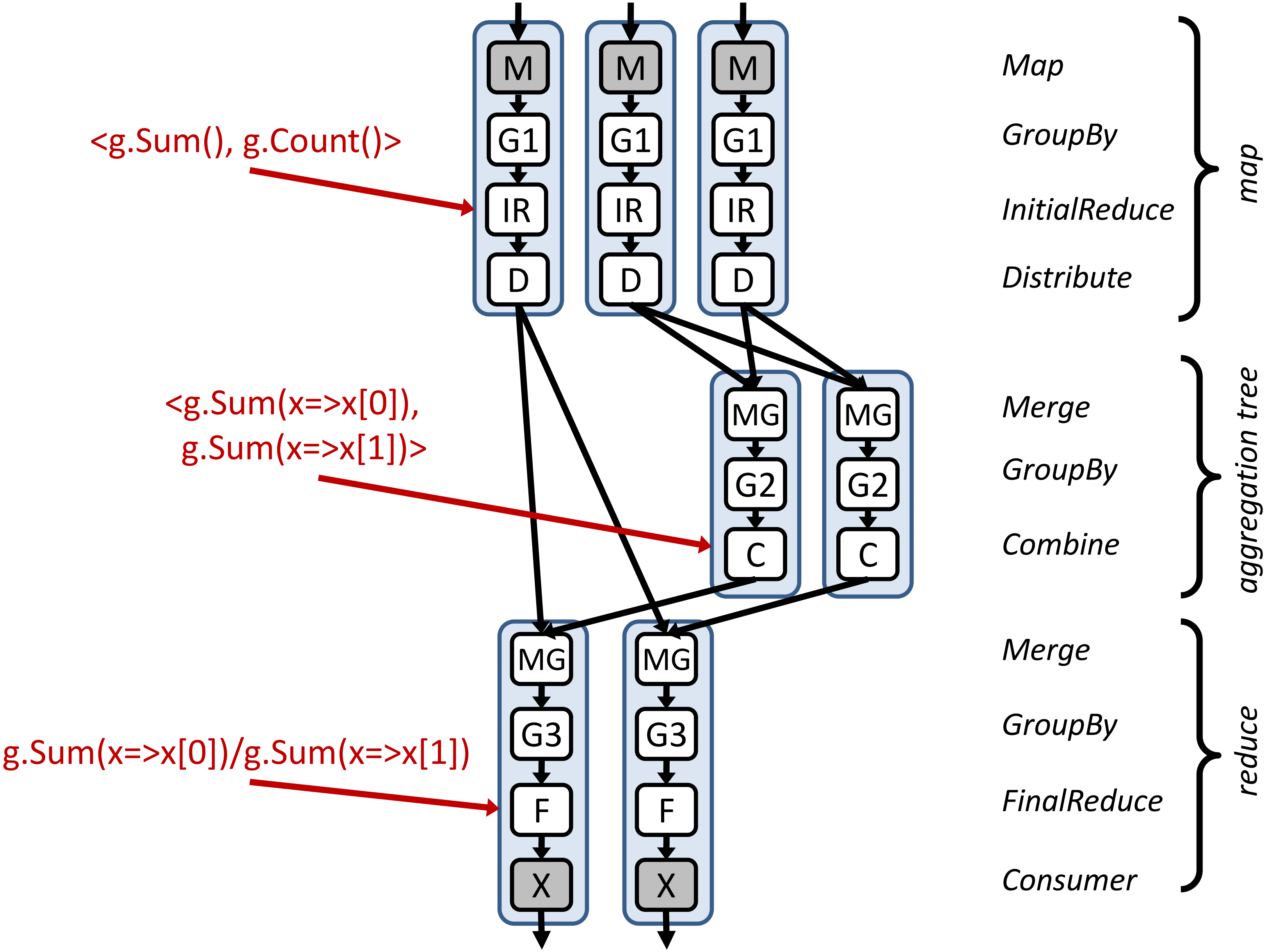
- Group a sequence of numbers into groups and compute the average for each group

```
source = <sequence of numbers>
groups = source.GroupBy(keySelector);
reduce = groups.Select(g => g.Sum()/g.Count());
```

Naïve Execution Plan



Execution Plan Using Partial Aggregation



Distributed Aggregation in DryadLINQ

- The programmer simply writes:

```
source = <sequence of integers>
groups = source.GroupBy(keySelector);
reduce = groups.Select(g => g.Sum()/g.Count());
```

- The system takes care of the rest
  - Generate an efficient execution plan
  - Provide efficient, reliable execution

Outline

- Programming interfaces
- Implementations
- Evaluations
- Discussion and conclusions

Decomposable Functions

- Roughly, a function H is decomposable if it can be expressed as composition of two functions IR and C such that
  - IR is commutative
  - C is commutative and associative
- Some decomposable functions
  - Sum: IR = Sum, C = Sum
  - Count: IR = Count, C = Sum
  - OrderBy.Take: IR = OrderBy.Take,
C = SelectMany.OrderBy.Take

Two Key Questions

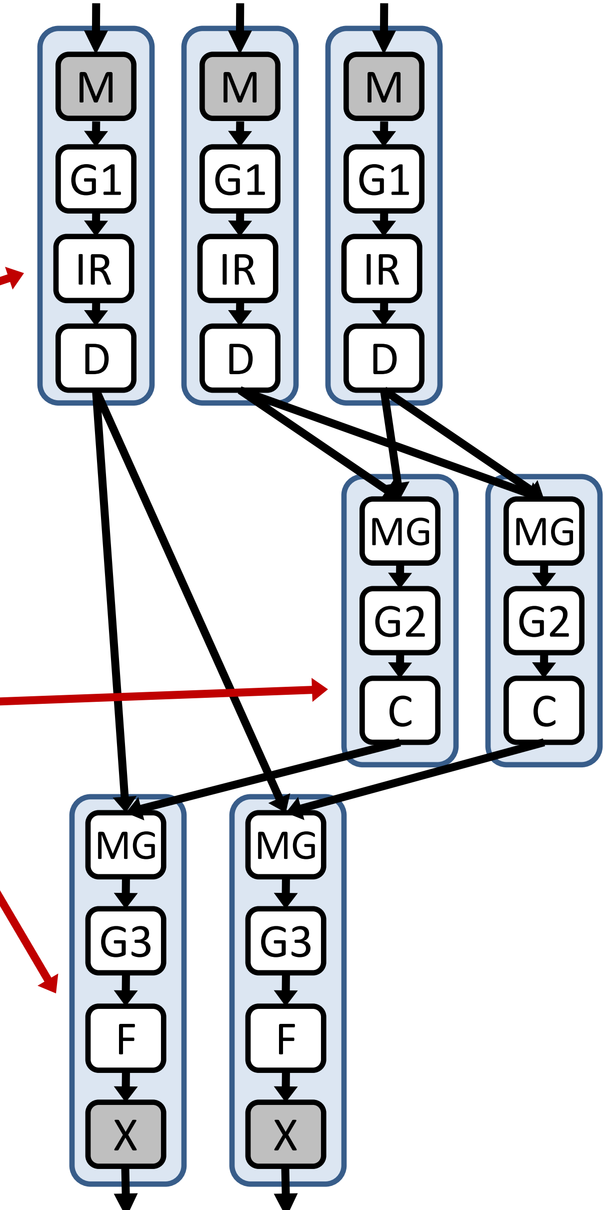
- How do we decompose a function?
  - Two interfaces: iterator and accumulator
  - Choice of interfaces can have significant impact on performance
- How do we deal with user-defined functions?
  - Try to infer automatically
  - Provide a good annotation mechanism

Iterator Interface in DryadLINQ

```
[Decomposable("InitialReduce", "Combine")]
public static IntPair SumAndCount(IEnumerable<int> g) {
    return new IntPair(g.Sum(), g.Count());
}

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());
}
```



Iterator Interface in Hadoop

```
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([...]);
  } } }

static public class Intermed extends EvalFunc<Tuple> {
 @Override public void exec(Tuple input, Tuple output)
    throws IOException {
  combine(input.getBagField(0), output); }  }

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;
}
```

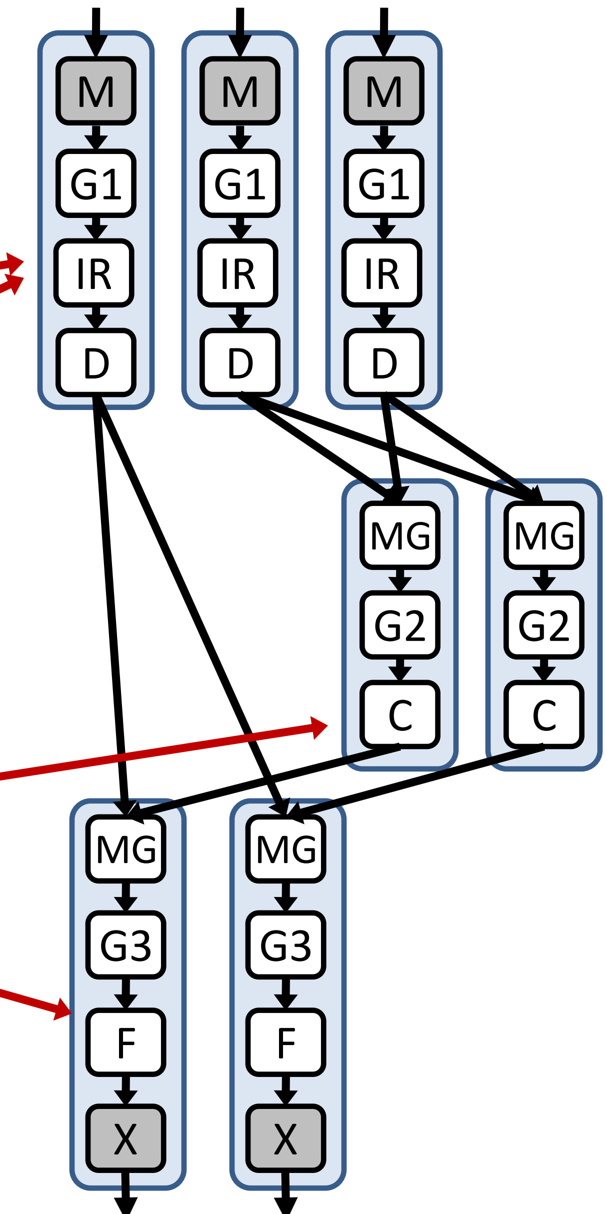
Accumulator Interface in DryadLINQ

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

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;
}
```



Accumulator Interface in Oracle

```
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;
```

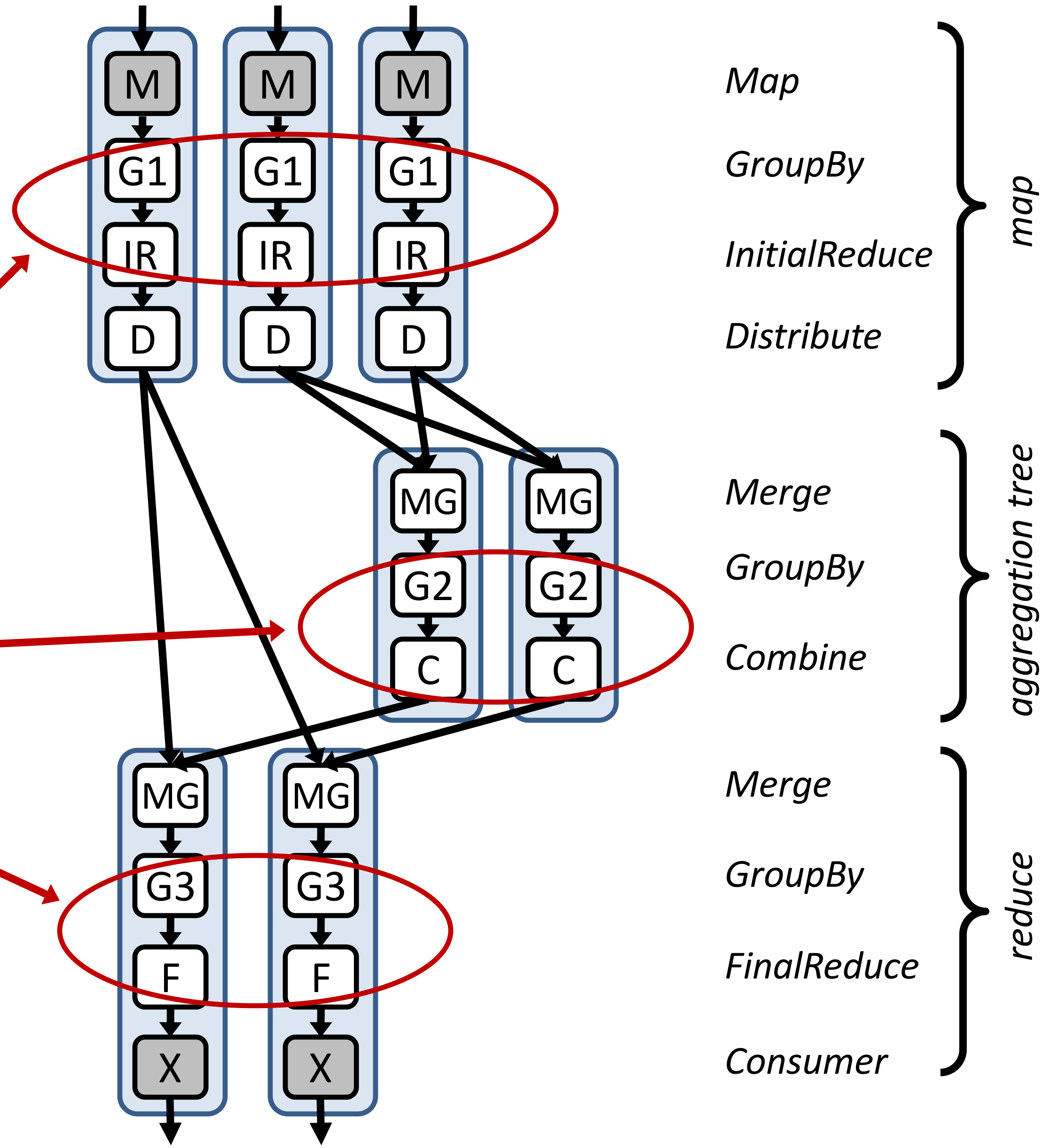
Decomposable Reducers

- Recall our GroupBy-Aggregate:

  groups = source.**GroupBy**(keySelector);
  reduce = groups.**SelectMany**(reducer);

- Intuitively, reducer is decomposable if every leaf function call is of form H(g) for some decomposable function H
- Some decomposable reducers
  - Average: g.Sum()/g.Count()
  - SDV: Sqrt(g.Sum(x=>x*x)-g.Sum()*g.Sum())
  - $F(H_1(g), H_2(g))$, if H_1 and H_2 are decomposable

Implementation



Aggregation steps:

- G1+IR
- G2+C
- G3+F

Implementations

- Key considerations
  - Data reduction of the partial aggregation stages
  - Pipelining with upstream/downstream computations
  - Memory consumption
  - Multithreading to take advantage of multicore machines
- Six aggregation strategies
  - Iterator-based: FullSort, PartialSort, FullHash, PartialHash
  - Accumulator-based: FullHash, PartialHash

Iterator PartialSort

- G1+IR and G2+C
  - Keep only a fixed number of chunks in memory
  - Chunks are processed in parallel: sorted, grouped, reduced by IR or C, and emitted
- G3+F
  - Read the entire input into memory, perform a parallel sort, and apply F to each group
- Observations
  - G1+IR can always be pipelined with upstream
  - G3+F can often be pipelined with downstream
  - G1+IR may have poor data reduction
  - PartialSort is the closest to MapReduce

Accumulator FullHash

- G1+IR, G2+C, and G3+F
  - Build an in-memory parallel hash table: one accumulator object/key
  - Each input record is “accumulated” into its accumulator object, and then discarded
  - Output the hash table when all records are processed
- Observations
  - Optimal data reduction for G1+IR
  - Memory usage proportional to the number of unique keys, not records
    - So, we by default enable upstream and downstream pipelining
  - Used by DB2 and Oracle

Evaluation

- Example applications
  - **WordStats** computes word statistics in a corpus of documents (140M docs, 1TB total size)
  - **TopDocs** computes word popularity for each unique word (140M docs, 1TB total size)
  - **PageRank** performs PageRank on a web graph (940M web pages, 700GB total size)
- Experiments were performed on a 240-node Windows cluster
  - 8 racks, 30 machines per rack

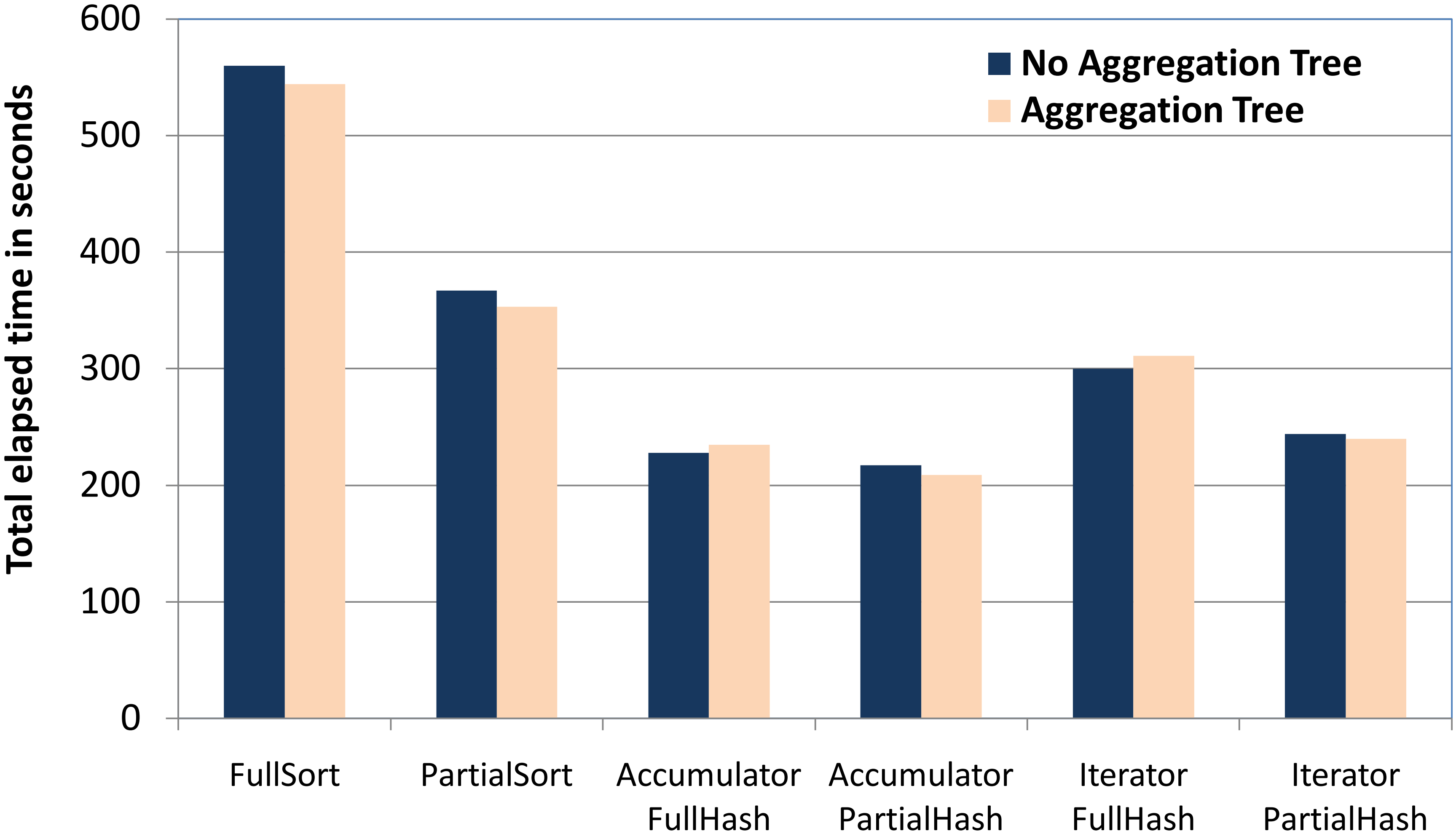
Example: WordStats

```
var docs = PartitionedTable.Get<Doc>("dfs://docs.pt");

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());

wordStats.ToPartitionedTable("dfs://result.pt");
```

WordStats Performance



WordStats Performance

- Comparison with baseline (no partial aggregation)
  - Baseline: 900 seconds
  - FullSort: 560 seconds
  - Mainly due to additional disk and network IO
- Comparison with MapReduce
  - Simulated MapReduce in DryadLINQ
    - 16000 mappers and 236 reducers
    - Machine-level aggregation
  - MapReduce: 700 seconds
    - 3x slower than Accumulator PartialHash

WordStats Data Reduction

- The total data reduction is about 50x

Strategy	G1+IR	G2+C	G3+F
FullSort	11.7x	2.5x	1.8x
PartialSort	3.7x	7.3x	1.8x
AccFullHash	11.7x	2.5x	1.8x
AccPartialHash	4.6x	6.15x	1.85x
IterFullHash	11.7x	2.5x	1.8x
IterPartialHash	4.1x	6.6x	1.9x

- The partial strategies are less effective in G1+IR
  - Always use G2+C in this case

Discussion and Conclusions

- Programming Interfaces
  - Have big impact on the actual performance
    - Accumulator interface was the winner
  - DryadLINQ offers better interfaces than Hadoop and databases
    - Better integration with the existing programming languages and their type systems
    - Enable compositions of decomposable functions
  - Iterator is somewhat easier to program with
    - Adopted by .NET and LINQ
    - Adopted by MapReduce/Hadoop

Discussion and Conclusions

- Implementations
  - Accumulator-FullHash was the winner
    - Database folks got it right here ☺
    - PartialSort (closest to MapReduce) was the second worst strategy
  - Need to choose between various optimizations
    - Rack-level aggregation?
    - FullHash or PartialHash?
    - Pipelining or not?
    - …

Discussion and Conclusions

- GroupBy-Aggregate is an extremely important primitive for data-parallel computing
- We need to get its programming model right!

Dryad/DryadLINQ Availability

- Freely available for academic use
  - http://connect.microsoft.com
  - Dryad in binary, DryadLINQ in source
  - Will release Dryad source in the future
- Will release to Microsoft commercial partners
  - Free, but no product support

Software Stack