Jet: An embedded DSL for Distributed Data Parallel Computing

Master Thesis Project
EPFL 2012
Stefan Ackermann (ETHZ)

Supervisors:
MSc. Vojin Jovanovic (EPFL)
Prof. Martin Odersky (EPFL)

Intro: Big Data

- Huge data sets: Order of terabytes
- Data does not fit on one machine
- Variety of input formats

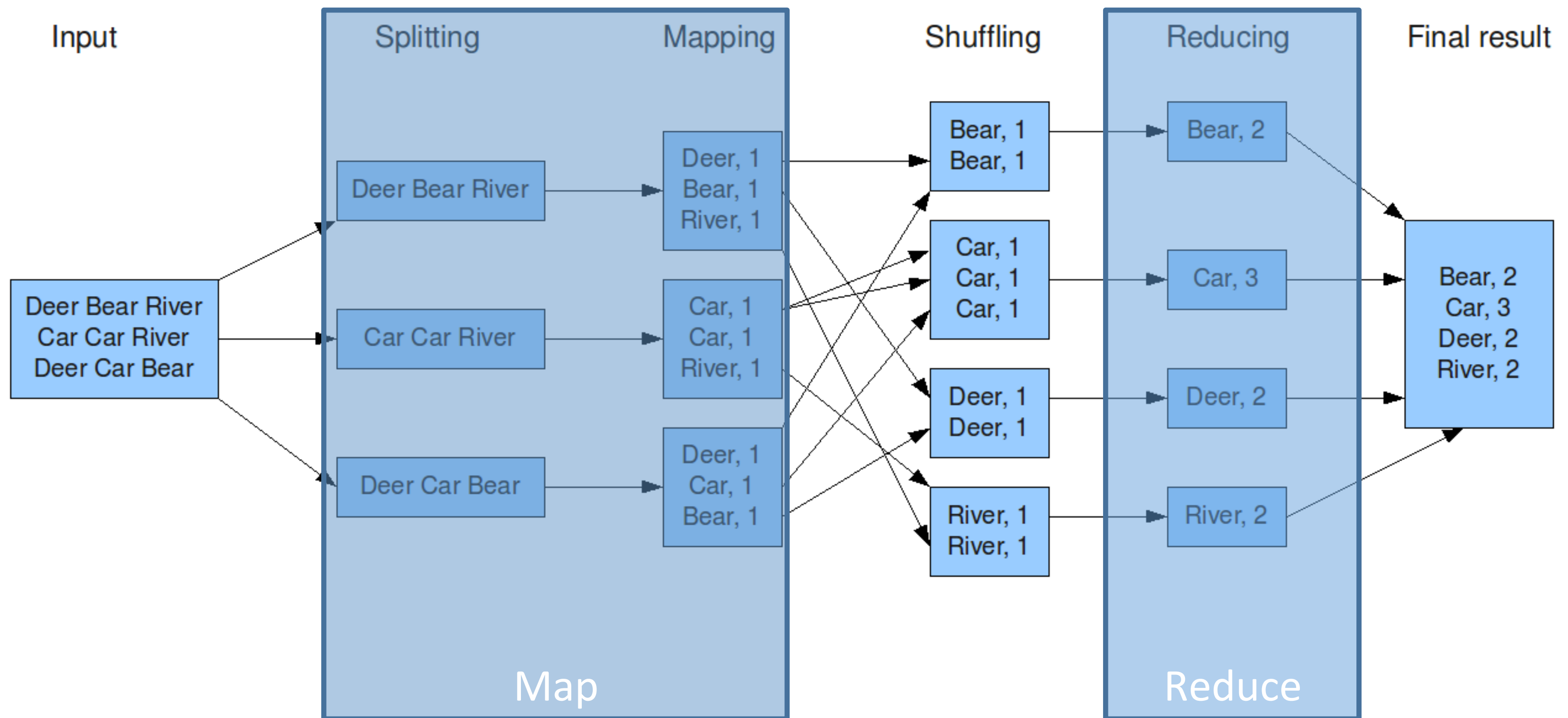
=> Databases not suitable

- Processing on commodity hardware
- Fault tolerant computations

Intro: Big Data in industry

- Pat Gelsinger (CEO of EMC): Big Data is a business of 70 Billion $ up, with an annual growth of 15%
- Big internet companies are all invested: Google (MapReduce, FlumeJava, …), Facebook (Hive), Yahoo! (Pig) and Microsoft / Bing (Dryad, DryadLINQ, Scope)

Intro: MapReduce



Intro: Hadoop

- Opensource MapReduce implementation
- Scalable
- Fault tolerant
- But:
  - Low level. Just one map and one reduce phase per Job. No joins. No sorting. Needs serialization
  - Wordcount: 58 lines

Intro: Pig

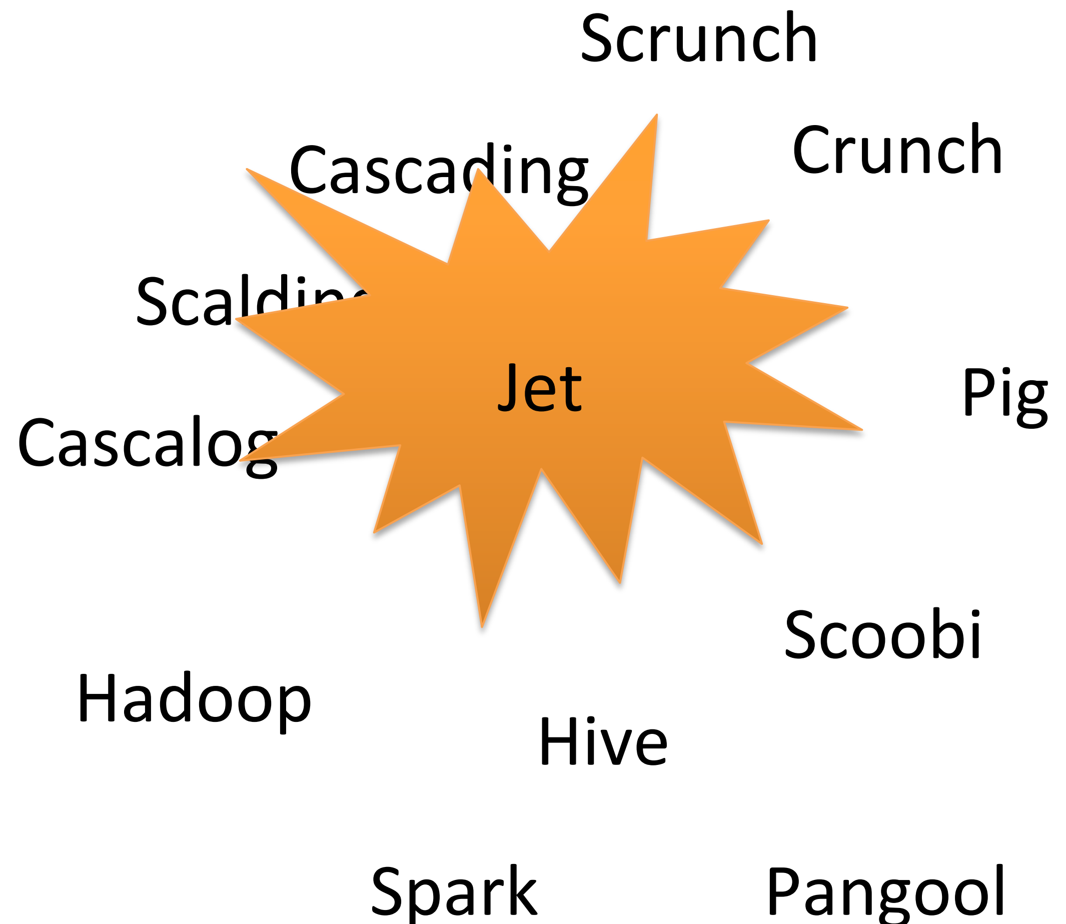
- DSL for Hadoop
- Has SQL like syntax, with assignments
  - Joins, sorting, ...
- Performs relational optimizations
- Wordcount: 5 lines

Intro: Pig downsides

- I wanted a Wordcount using a different pattern to split on
  - 2 days of effort
  - needs external function (~ 100 lines of code)
- Pig Latin does not have functions or classes
- Pig Latin does not have loops
- User defined functions must be in other language and break optimizations

Intro: Frameworks

- High level
- Automatic Serialization
- Projection Insertion
- Iterative jobs
- Language Embedding
- Extensibility
- Code portability



Jet

Wordcount in Jet

```
DList("hdfs://..." + input)
  .flatMap(_.split("\\s"))
  .map(x => (x, 1))
  .groupByKey()
  .reduce(_ + _)
  .save("hdfs://..." + output)
```

User Defined Function in Jet

```
def parse(x: Rep[String]): Rep[String] = {
  x.trim().split("\\s+").apply(2)
}
```

Jet

- Applies compile time optimizations
- Extensible / Modular
- General: Loops, conditionals
- Portable: Compiles to Scala code for Crunch (Hadoop) and Spark
  - Some operations specific to one backend

Jet Modularity

- Code generation is completely separated from the optimizations
- Code generation is small: 400 Lines of code per backend
- Crunch backend: One week of effort

Outline

- Background
- Optimizations
- Evaluation
- Conclusion

Background: Frameworks

- All offer a collection like interface
- Hadoop
  - Crunch: Java based
  - Scoobi: Scala based
- Spark
  - Spark: Scala based
  - Inspired by Hadoop
  - Keeps objects in memory by default

Background: LMS

- Framework for writing DSL's
- Basis for Jet
- Deeply embedded in Scala
- Modular / Extensible
- Effects tracking
- Code generation for multiple languages (C, CUDA, Scala)

Background: LMS Optimizations

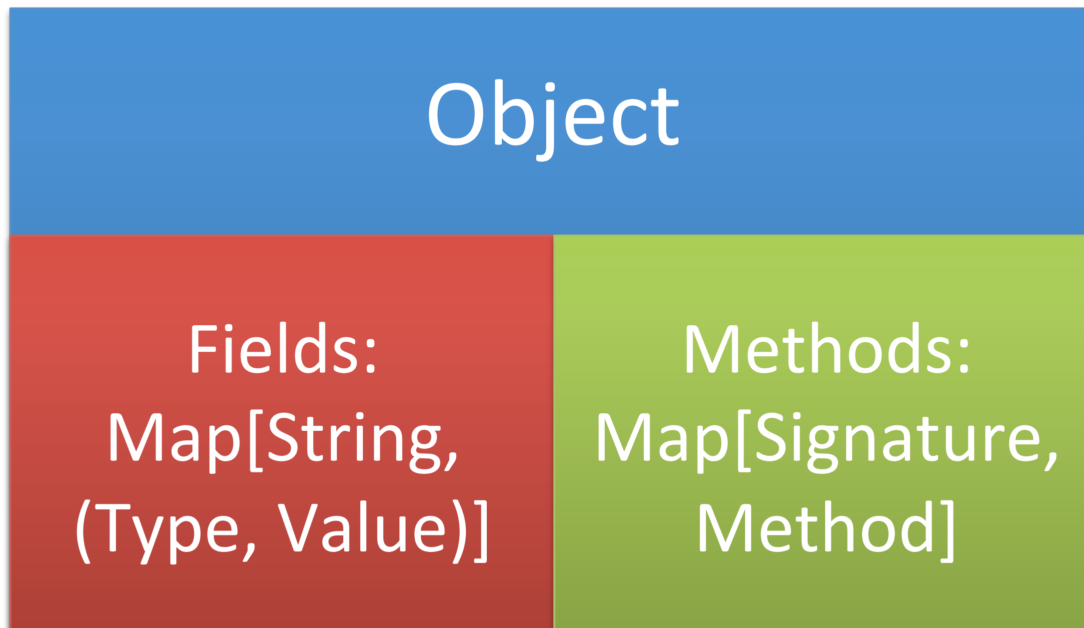
- Inline
  - Removes method calls
- Loop Fusion (vertical & horizontal)
- Code Motion
- Dead Code Elimination
- Structs

Background: Structs in LMS

- Assume: No subtyping
- With inlining

Idea:

Work with Fields directly



Background: Field Read Shortcut

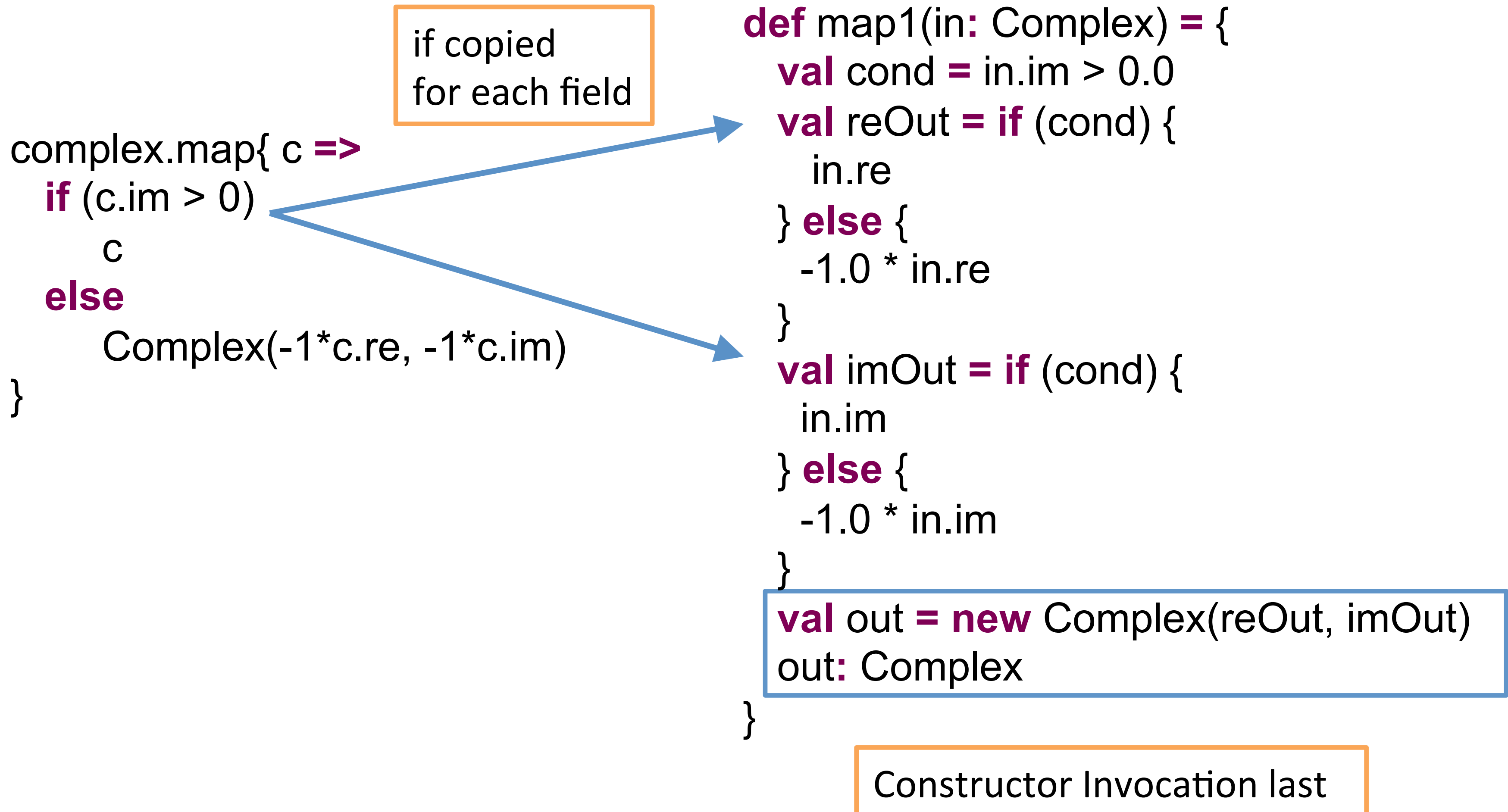
```
val complex = new Complex(re = 1, im = -1)
val re = complex.re
```

becomes

```
val re = 1
```

No object required => No object created

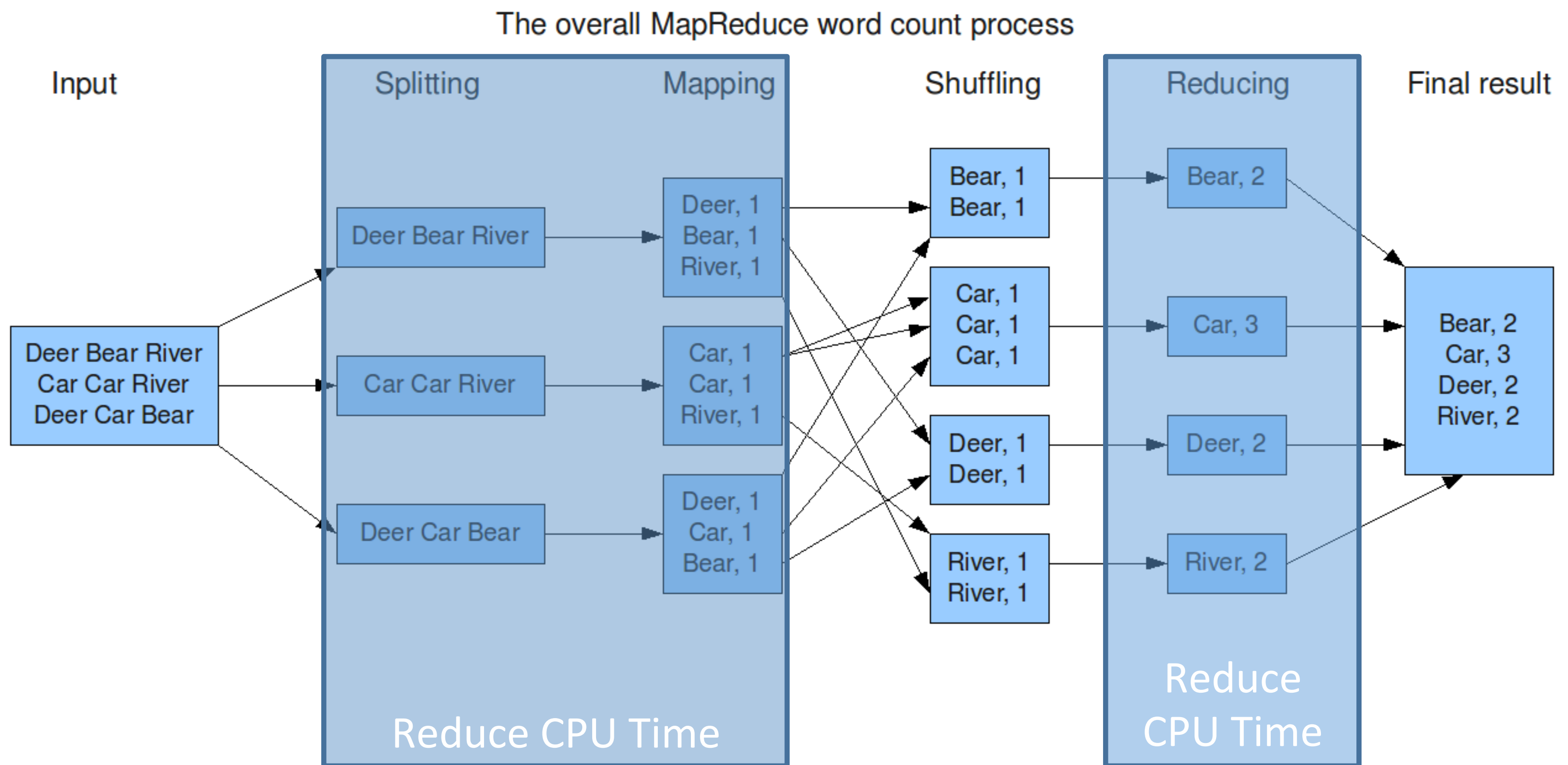
Background: Decomposition



Outline

- Background
- Optimizations
  - Code Motion
  - Loop Fusion
  - Projection Insertion
- Evaluation
- Conclusion

Optimizations in MapReduce



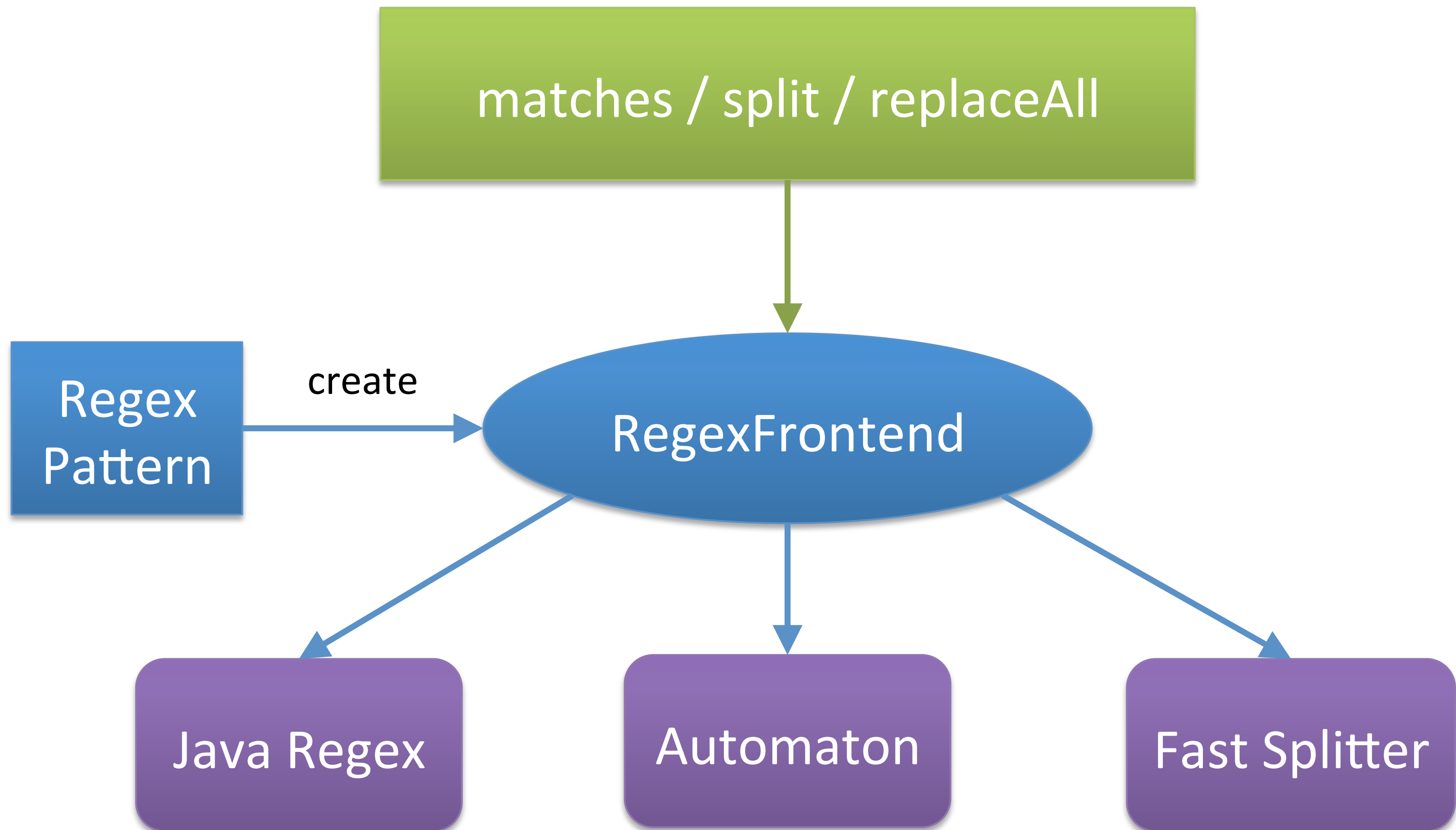
Optimizations: Code Motion

```
in.filter(s: String => s.matches("wiki"))
```

becomes

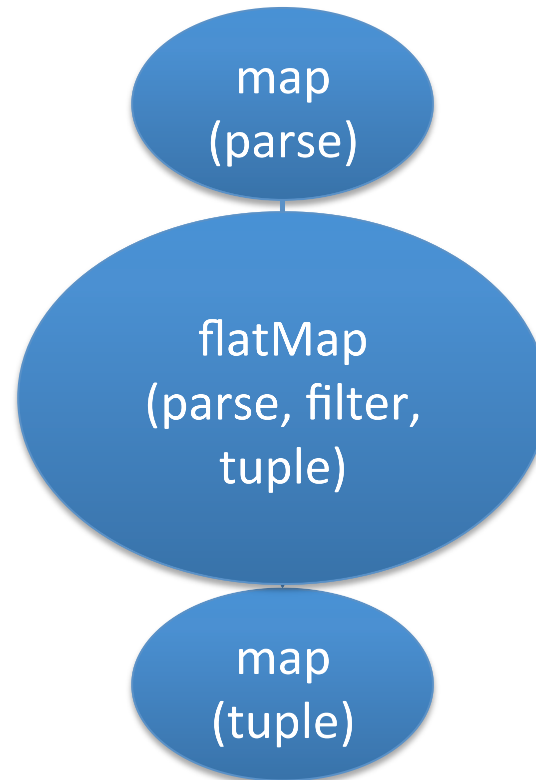
```
val pattern = Pattern.compile("wiki")
in.filter(s: String =>
  pattern.matcher(s).matches()
)
```

Optimizations: Regular Expressions



Optimizations: Loop Fusion

String

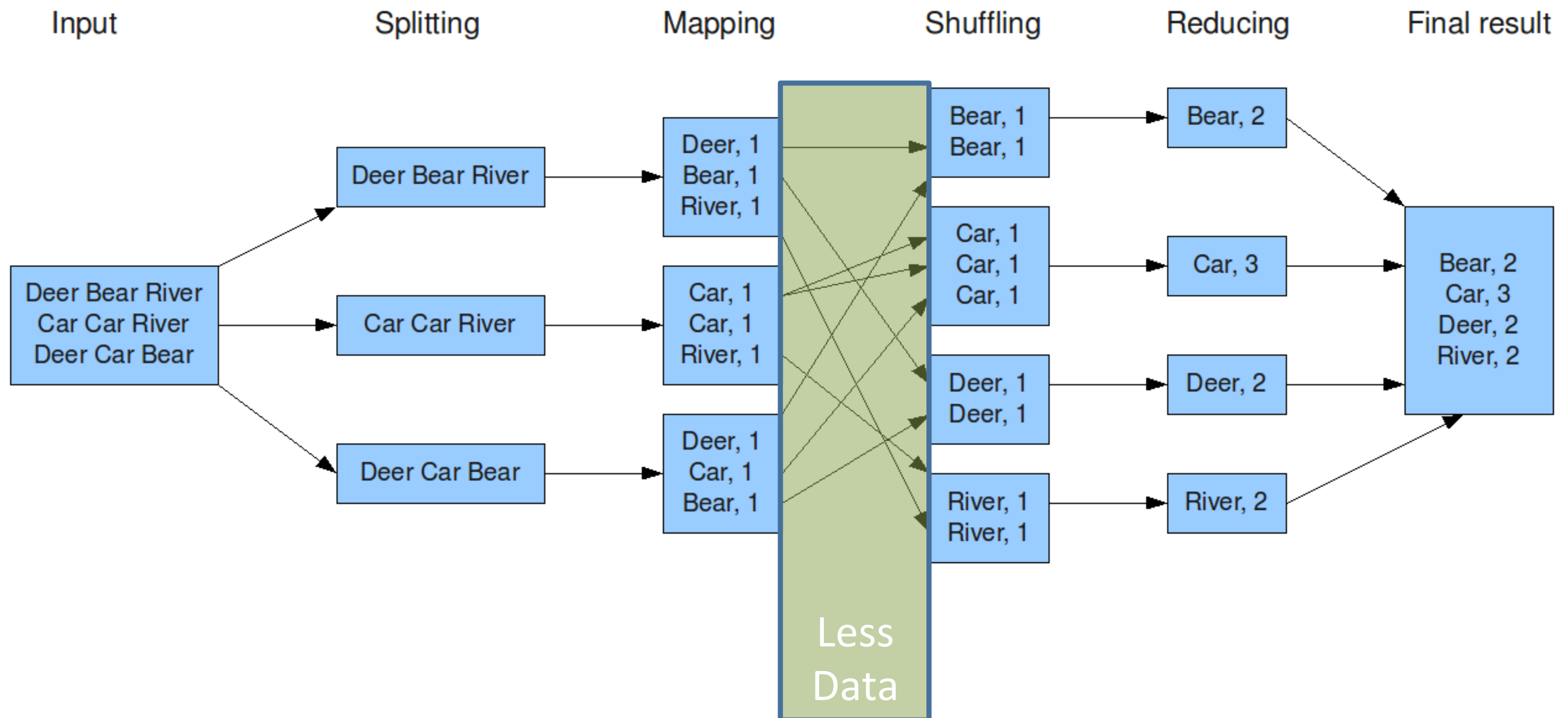


LogEvent

LogEvent

(Long, LogEvent)

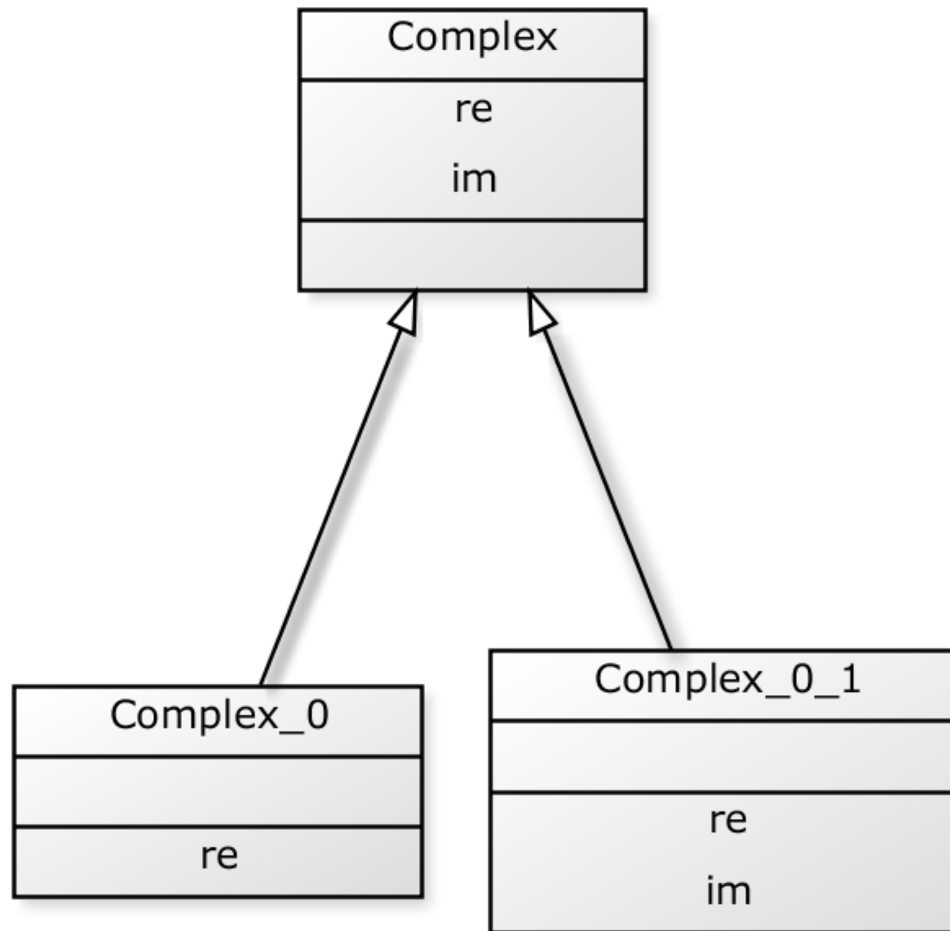
Optimizations in MapReduce



Projection Insertion: Goals

- Remove unneeded fields
- Reduce network traffic & disk writes
- Reduce CPU time, only parse necessary fields
- Spark: Reduce memory usage

Projection Insertion: Classes



Projection Insertion

```
def map1(in: Complex) = {
  val cond = in.im > 0.0
  val reOut = if (cond) {
     in.re
  } else {
    -1.0 * in.re
  }
  val imOut = if (cond) {
    in.im
  } else {
    -1.0 * in.im
  }
  val out = new Complex_0_1(reOut, imOut)
  out: Complex
}
```

We know: Only field «re» is needed afterwards.

```
def project(in: Complex) = {
      Complex_0(in.re)
}
```

Projection Insertion: Step 1

```
def map1(in: Complex) = {
  val cond = in.im > 0.0
  val reOut = if (cond) {
     in.re
  } else {
    -1.0 * in.re
  }
  val imOut = if (cond) {
    in.im
  } else {
    -1.0 * in.im
  }
  val out = new Complex_0_1(reOut, imOut)
  out: Complex
}

def project(in: Complex) = {
   Complex_0(in.re)
}
```

```
def map1(in: Complex) = {
   val cond = in.im > 0.0
   val reOut = if (cond) {
     in.re
   } else {
    -1.0 * in.re
   }
   val imOut = if (cond) {
    in.im
   } else {
    -1.0 * in.im
   }
   val out = new Complex_0_1(reOut, imOut)
   Complex_0(out.re)
}
```

Projection Insertion: Step 2

```
def map1(in: Complex) = {
    val cond = in.im > 0.0
    val reOut = if (cond) {
      in.re
    } else {
      -1.0 * in.re
    }
    val imOut = if (cond) {
      in.im
    } else {
      -1.0 * in.im
    }
    val out = new Complex_0_1(reOut, imOut)
    Complex_0(out.re)
  }
```

```
def map1(in: Complex) = {
   val cond = in.im > 0.0
   val reOut = if (cond) {
     in.re
   } else {
     -1.0 * in.re
   }
   val imOut = if (cond) {
     in.im
   } else {
     -1.0 * in.im
   }
   val out = new Complex_0_1(reOut, imOut)
   Complex_0(reOut)
}
```

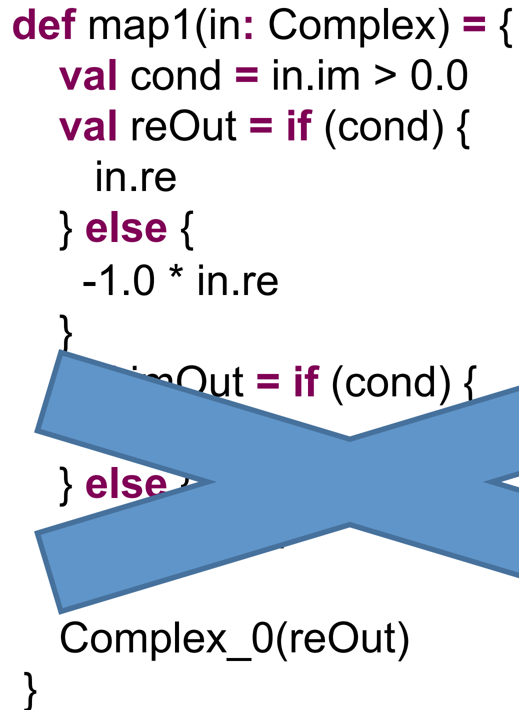
Projection Insertion: Step 3

```
def map1(in: Complex) = {
  val cond = in.im > 0.0
  val reOut = if (cond) {
    in.re
  } else {
   -1.0 * in.re
  }
  val imOut = if (cond) {
   in.im
  } else {
   -1.0 * in.im
  }
  val out = new Complex_0_1(reOut, imOut)
  Complex_0(reOut)
}
```

```
def map1(in: Complex) = {
  val cond = in.im > 0.0
  val reOut = if (cond) {
    in.re
  } else {
   -1.0 * in.re
  }
  val imOut = if (cond) {
   in.im
  } else {
   -1.0 * in.im
  }
  Complex_0(reOut)
}
```

Dead Code Elimination

Projection Insertion: Step 4




```
def map1(in: Complex) = {
  val cond = in.im > 0.0
  val reOut = if (cond) {
    in.re
  } else {
   -1.0 * in.re
  }
  Complex_0(reOut)
}
```

Dead Code Elimination

Projection Insertion: Analysis

```
def project(in: Complex) = {
      Complex_0(in.re)
}
```

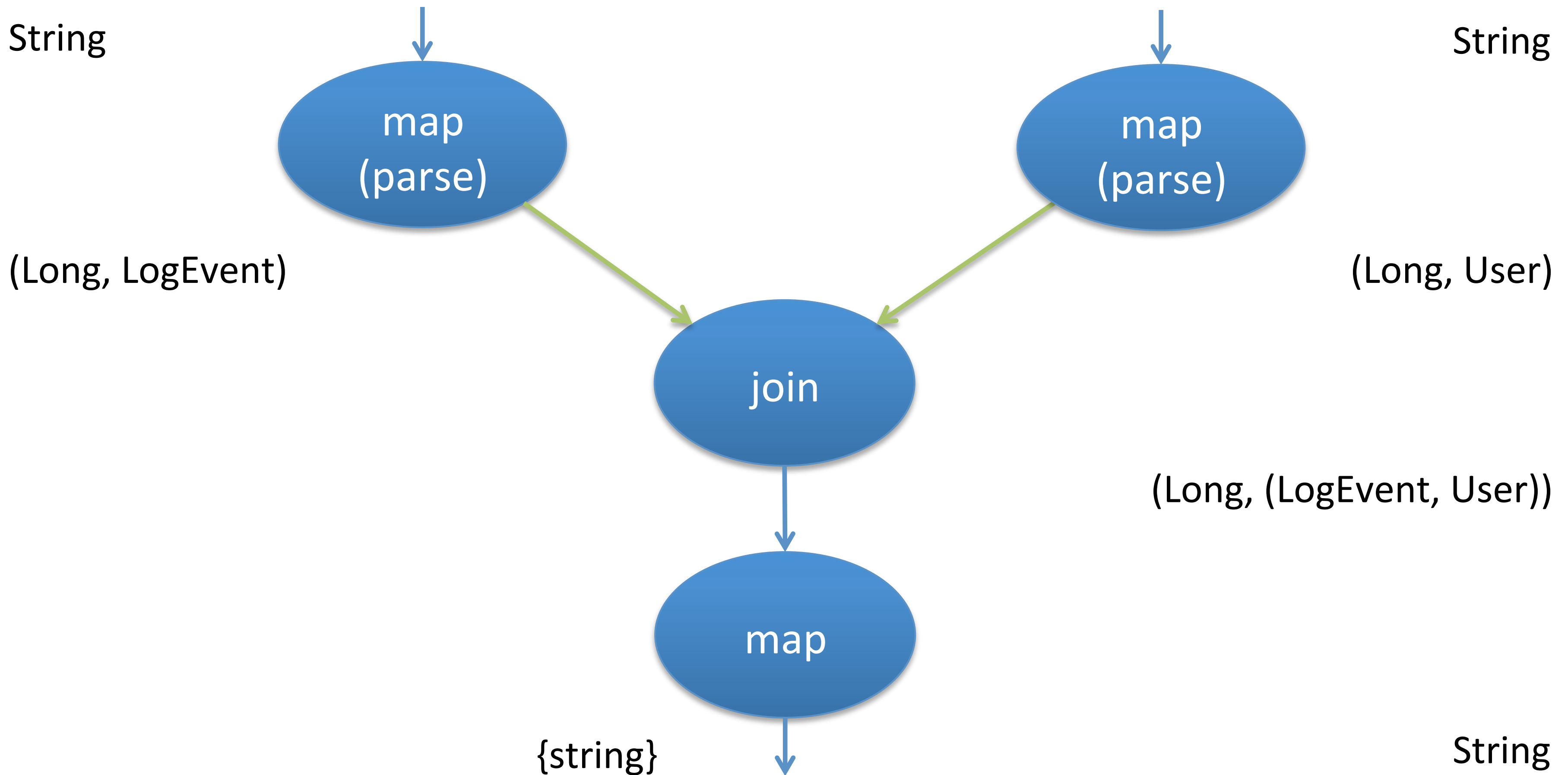
How do we know which fields to keep?

Projection Insertion: Analysis

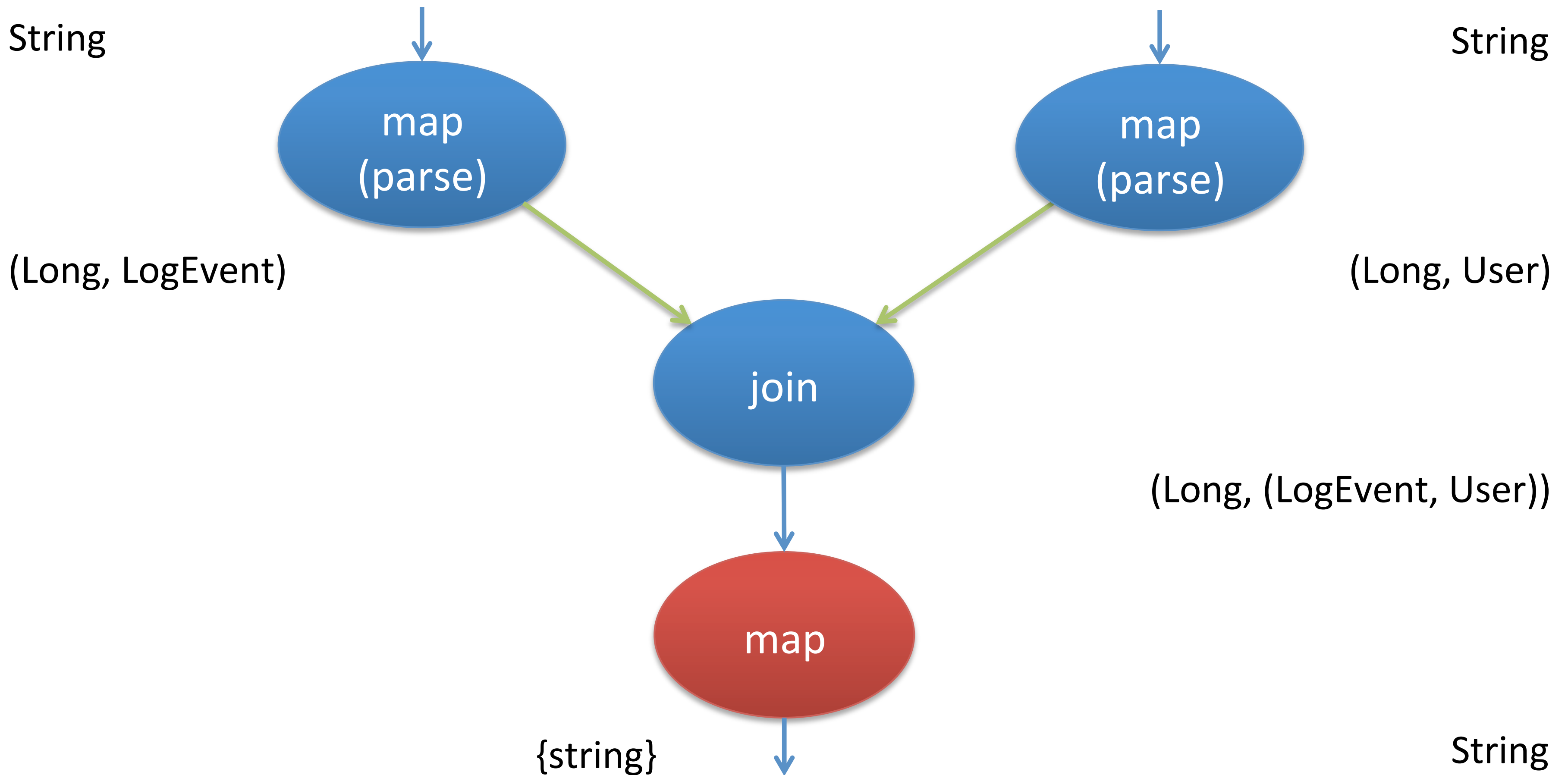
```
def map1(in: Complex) = {
  val cond = in.im > 0.0
  val reOut = if (cond) {
    in.re
  } else {
    -1.0 * in.re
  }
  Complex_0(reOut)
}
```

{complex.re, complex.im}

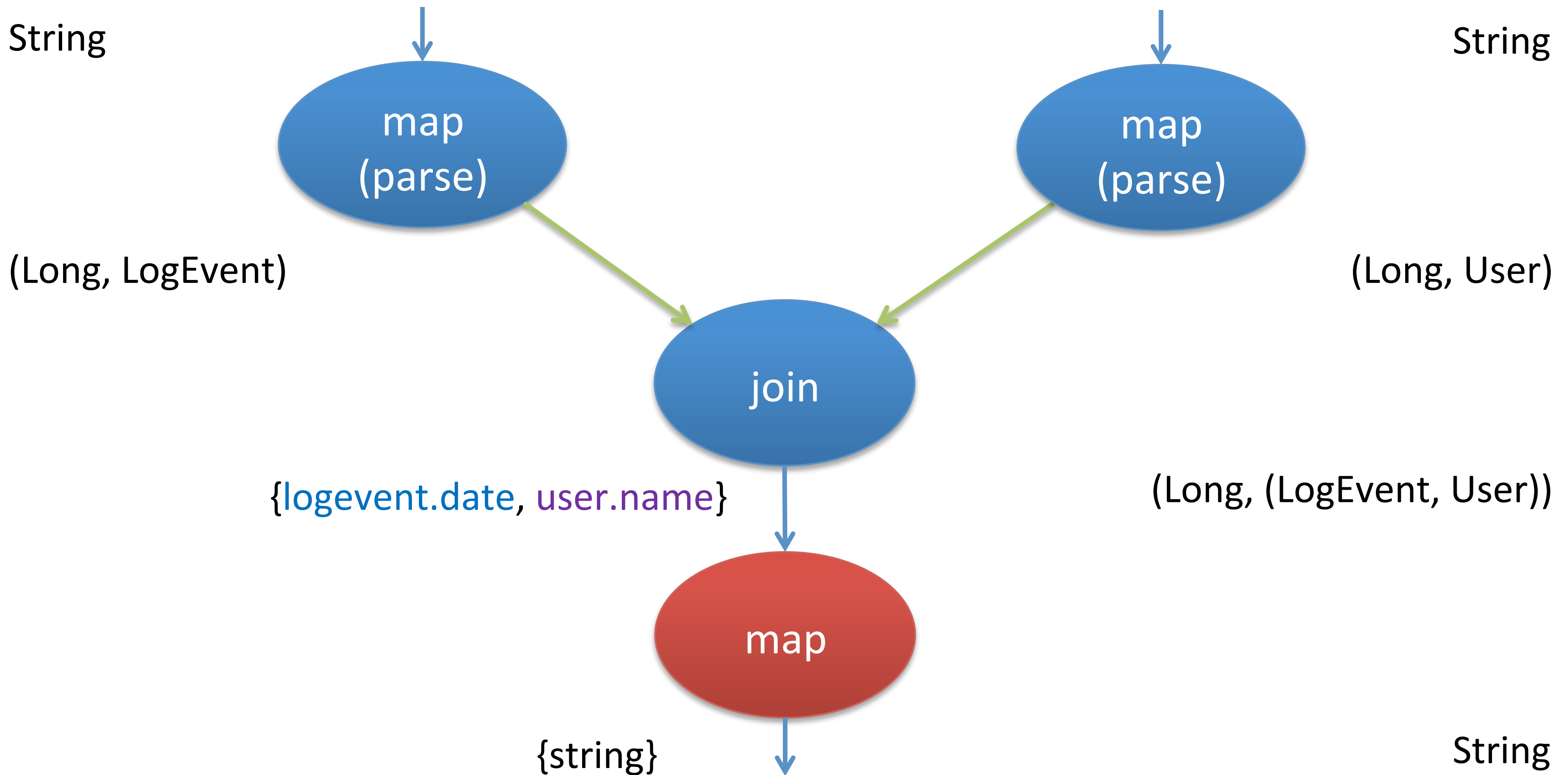
Projection Insertion: Propagation



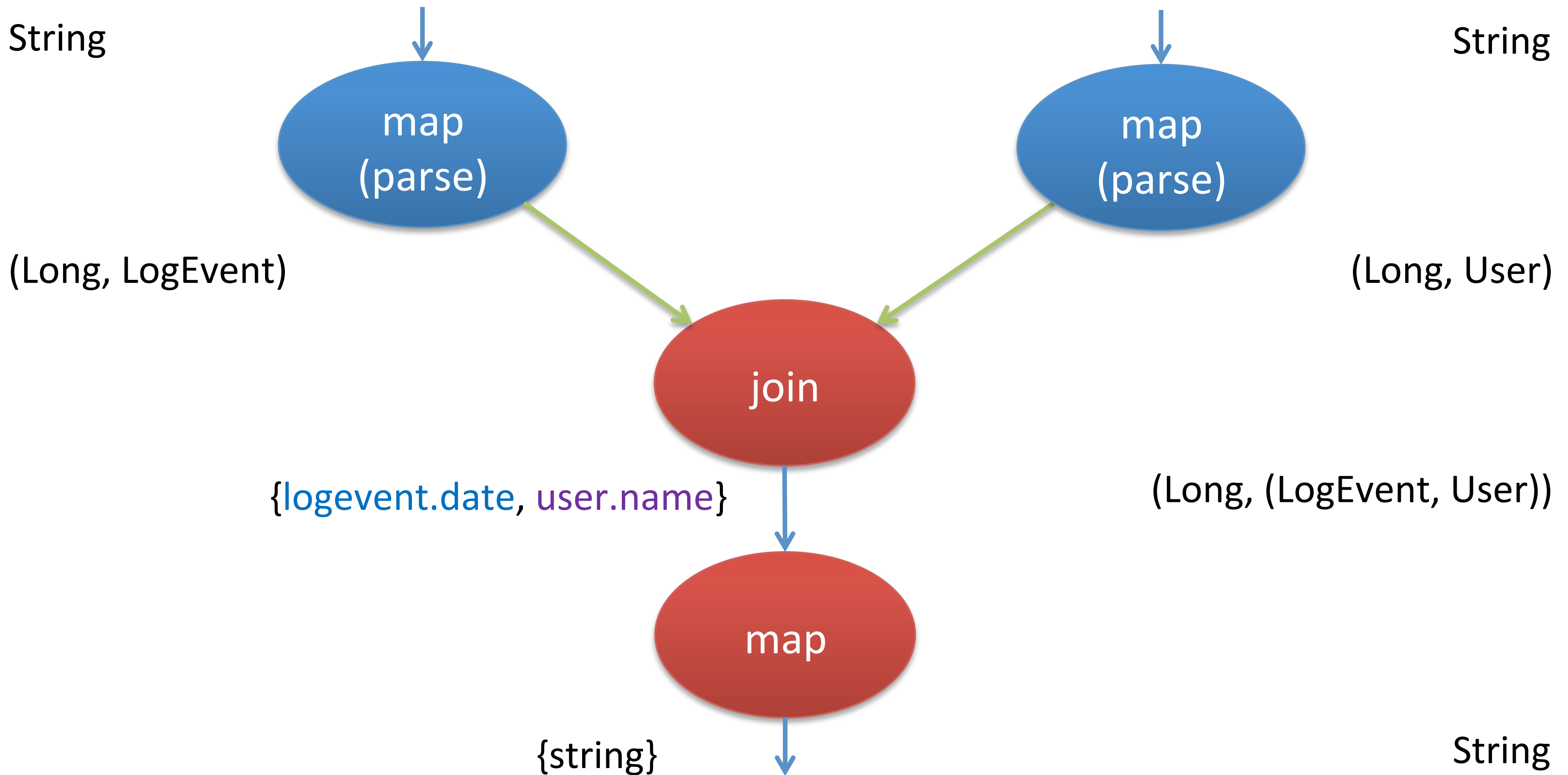
Projection Insertion: Propagation



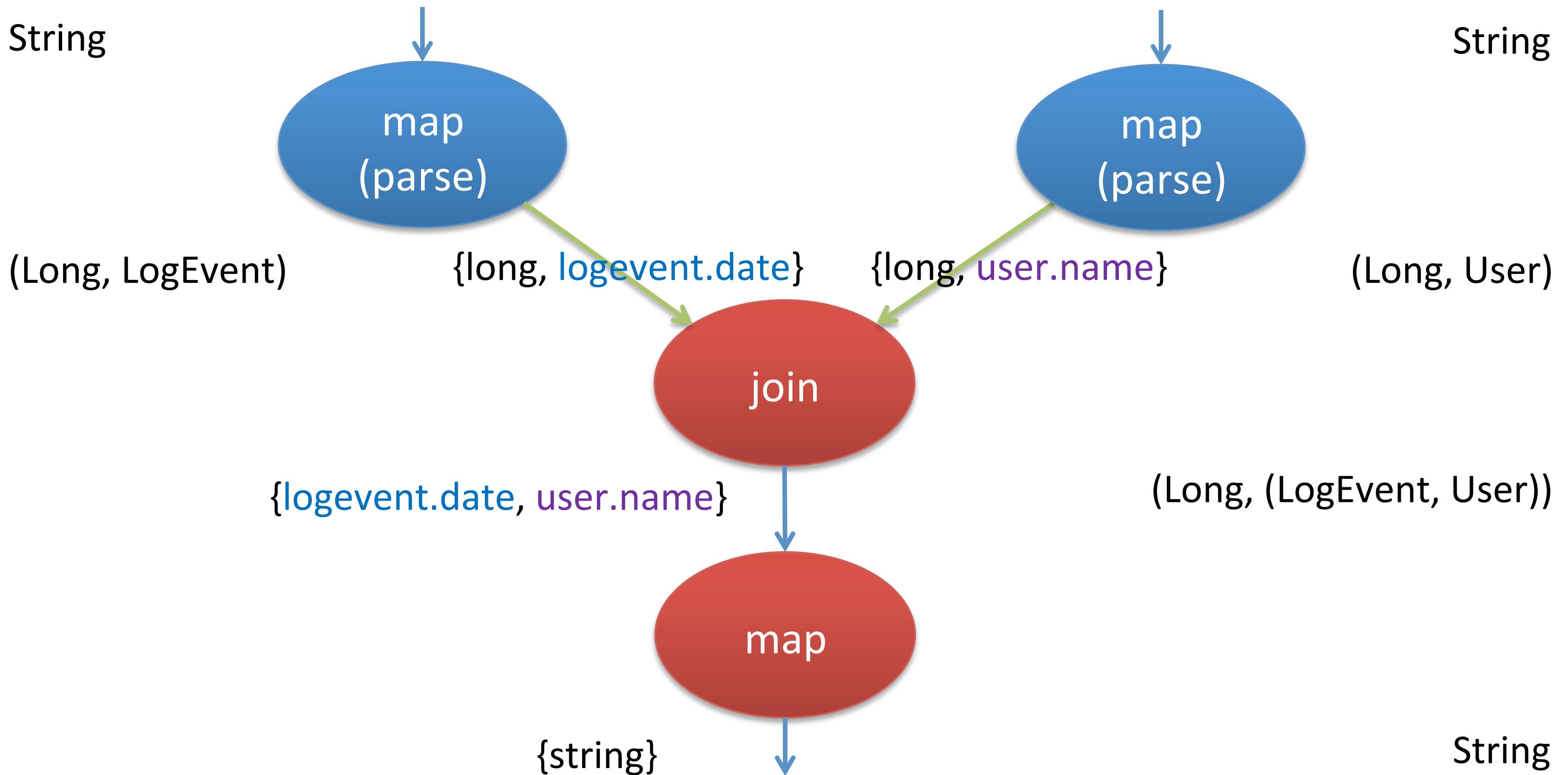
Projection Insertion: Propagation



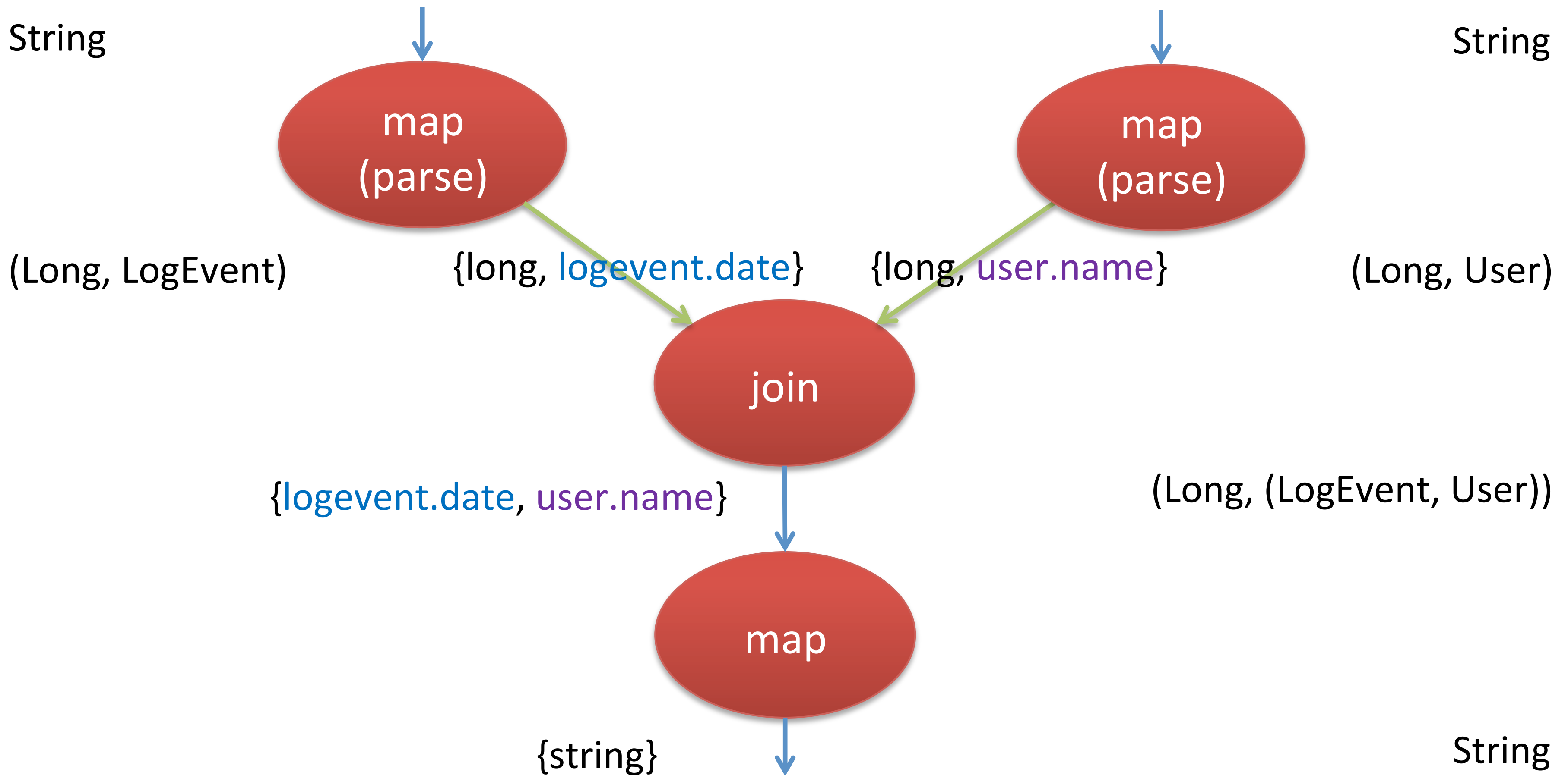
Projection Insertion: Propagation



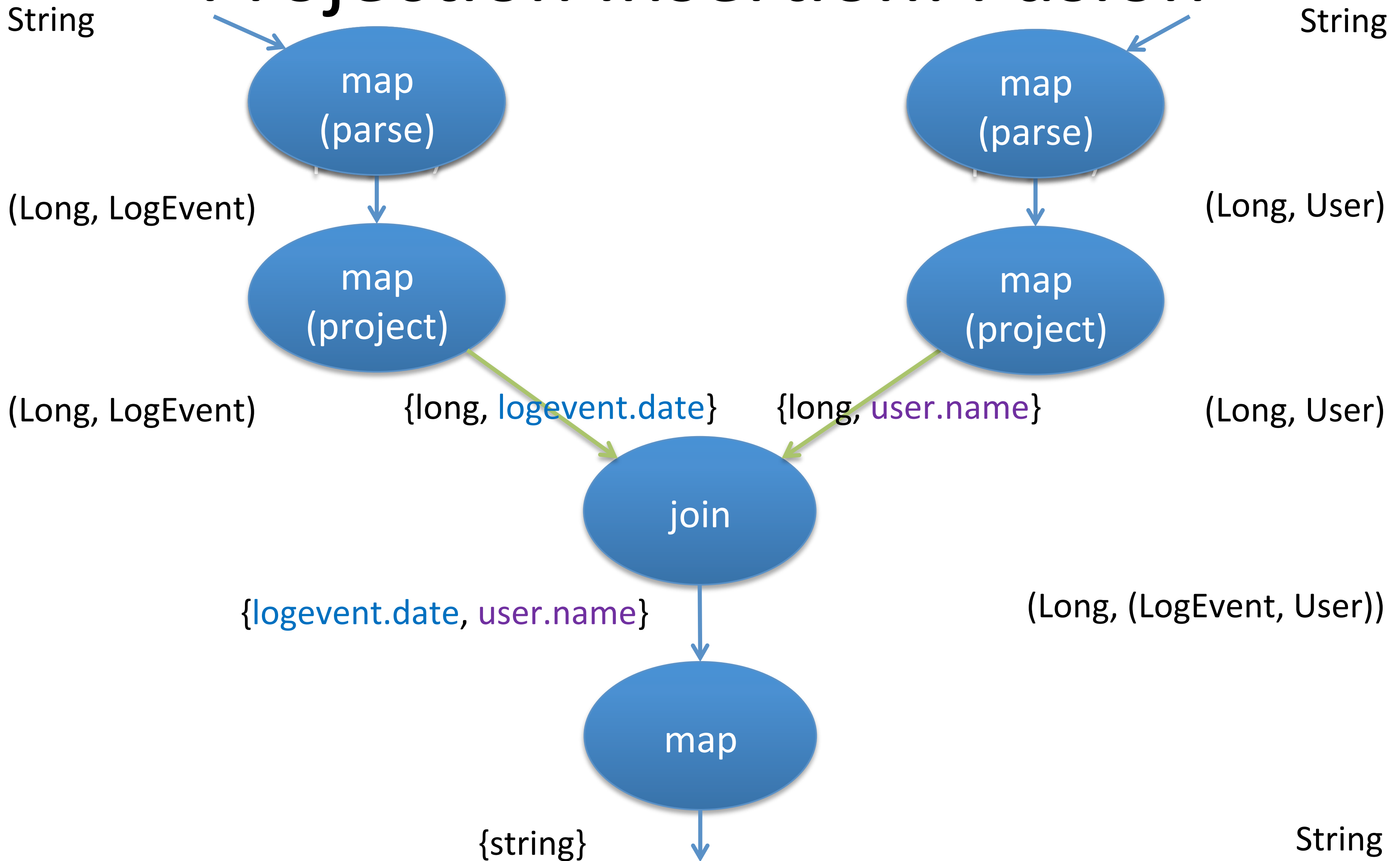
Projection Insertion: Propagation



Projection Insertion: Propagation

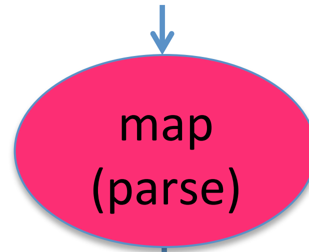


Projection Insertion: Fusion

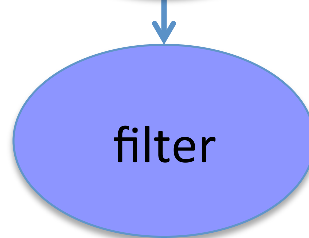


Optimizations: Mapper of TPCH Q12

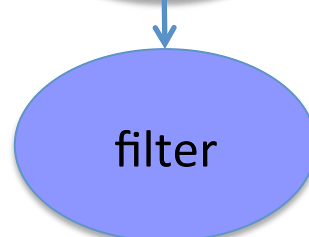
String

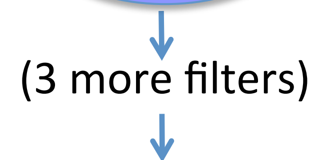


LineItem

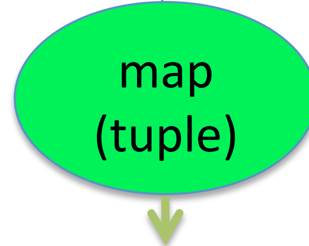


LineItem





LineItem



(Long, LineItem)

Optimizations: Mapper of TPCH Q12

Unoptimized

```
def x154(x10: (java.lang.String)) = {
  val x12 = x11.split(x10, 16);
  val x13 = x12(0);
  val x14 = x13.toInt;
  val x15 = x12(1);
  val x16 = x15.toInt;
  val x17 = x12(2);
  val x18 = x17.toInt;
  val x19 = x12(3);
  val x20 = x19.toInt;
  val x21 = x12(4);
  val x22 = x21.toDouble;
  val x23 = x12(5);
  val x24 = x23.toDouble;
  val x25 = x12(6);
  val x26 = x25.toDouble;
  val x27 = x12(7);
  val x28 = x27.toDouble;
  val x29 = x12(8);
  val x30 = x29.charAt(0);
  val x31 = x12(9);
  val x32 = x31.charAt(0);
  val x33 = x12(10);
  val x34 = ch.epfl.distributed.datastruct.Date(x33);
  val x35 = x12(11);
  val x36 = ch.epfl.distributed.datastruct.Date(x35);
  val x37 = x12(12);
  val x38 = ch.epfl.distributed.datastruct.Date(x37);
  val x39 = x12(13);
  val x40 = x12(14);
  val x41 = x12(15);
  val x42 = new LineItem_0_1_2_3_4_5_6_7_8_9_10_11_12_13_14_15
        (x14, x16, x18, x20, x22, x24, x26, x28, x30, x32, x34, x36, x38, x39, x40, x41);
  x42: LineItem
}
val x155 = x9.map(x154);
def x156(x67: (LineItem)) = {
  val x68 = x67.l_shipmode;
  val x69 = x68 == x6;
  val x71 = if (x69) {
   true
  } else {
   val x70 = x68 == x7;
   x70
  }
  x71: Boolean
}
val x157 = x155.filter(x156);
def x158(x74: (LineItem)) = {
  val x75 = x74.l_receiptdate;
  val x76 = x5 <= x75;
  x76: Boolean
}
val x159 = x157.filter(x158);
def x160(x79: (LineItem)) = {
  val x80 = x79.l_shipdate;
  val x81 = x79.l_commitdate;
  val x82 = x80 < x81;
  x82: Boolean
}
val x161 = x159.filter(x160);
def x162(x85: (LineItem)) = {
  val x86 = x85.l_commitdate;
  val x87 = x85.l_receiptdate;
  val x88 = x86 < x87;
  x88: Boolean
}
val x163 = x161.filter(x162);
def x164(x91: (LineItem)) = {
  val x92 = x91.l_receiptdate;
  val x94 = x92 < x93;
  x94: Boolean
}
val x165 = x163.filter(x164);
def x166(x102: (LineItem)) = {
  val x103 = x102.l_orderkey;
  val x104 = (x103, x102);
  x104: scala.Tuple2[Int, LineItem]
}
val x167 = x165.map(x166);
```

Fields: 16 / 16

Parsing

Filtering

Creating Tuple

Outline

- Background
- Optimizations
- Evaluation
  - WordCount
  - TPCH Q12
  - KMeans
  - Jet vs Pig
- Conclusion

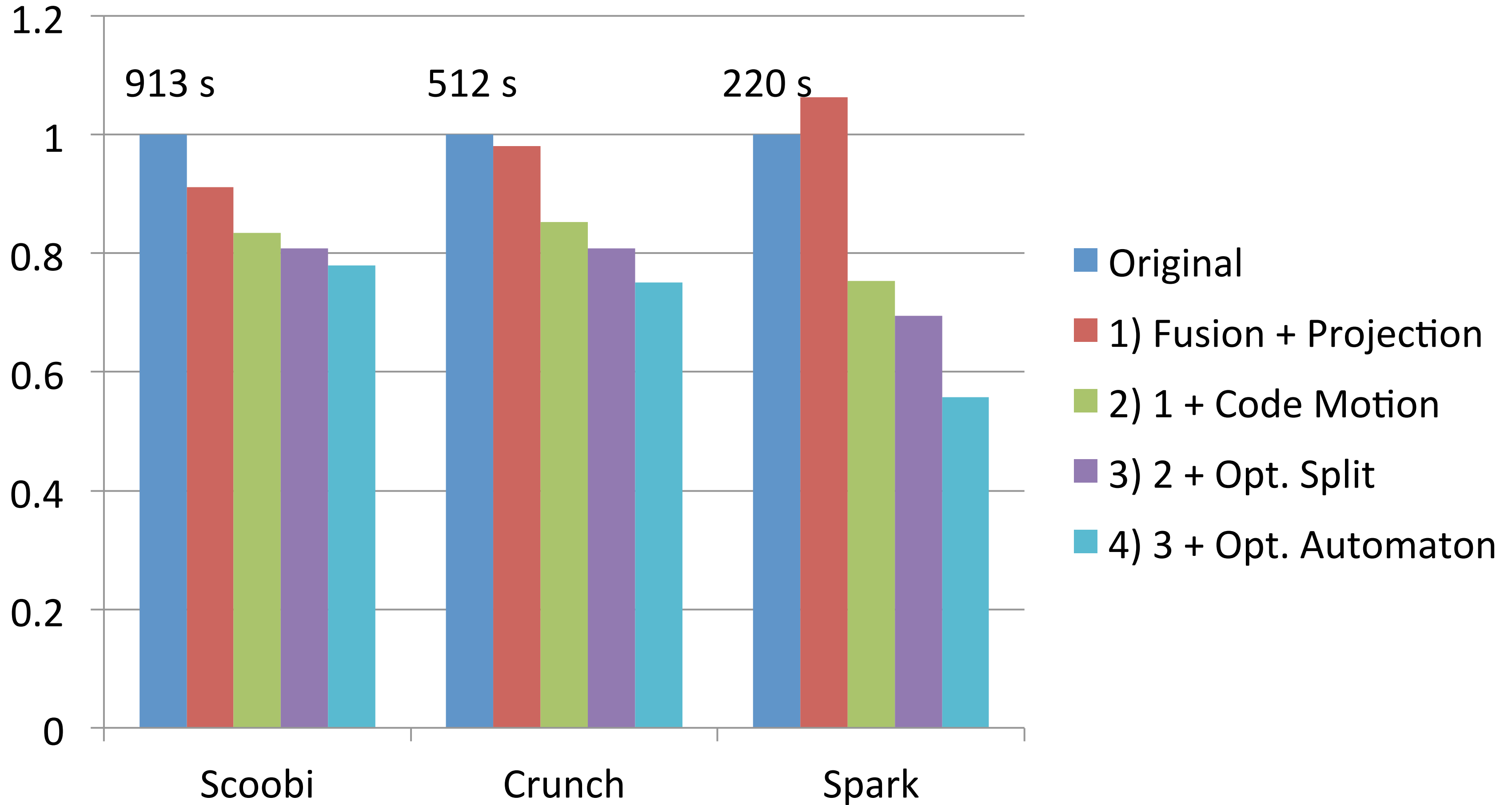
Results: Setup

- Amazon EC2 Cloud
- 21 EC2 m1.large nodes (1 master, 20 slaves)
  - 7.5 Gb Ram
  - 2 Cores
  - 2 Hard disks
  - Gbit connections

Results: Wordcount

- Program has only one map and one reduce phase
- Uses 5 regular expressions
- Input: 62 Gb Wikipedia articles

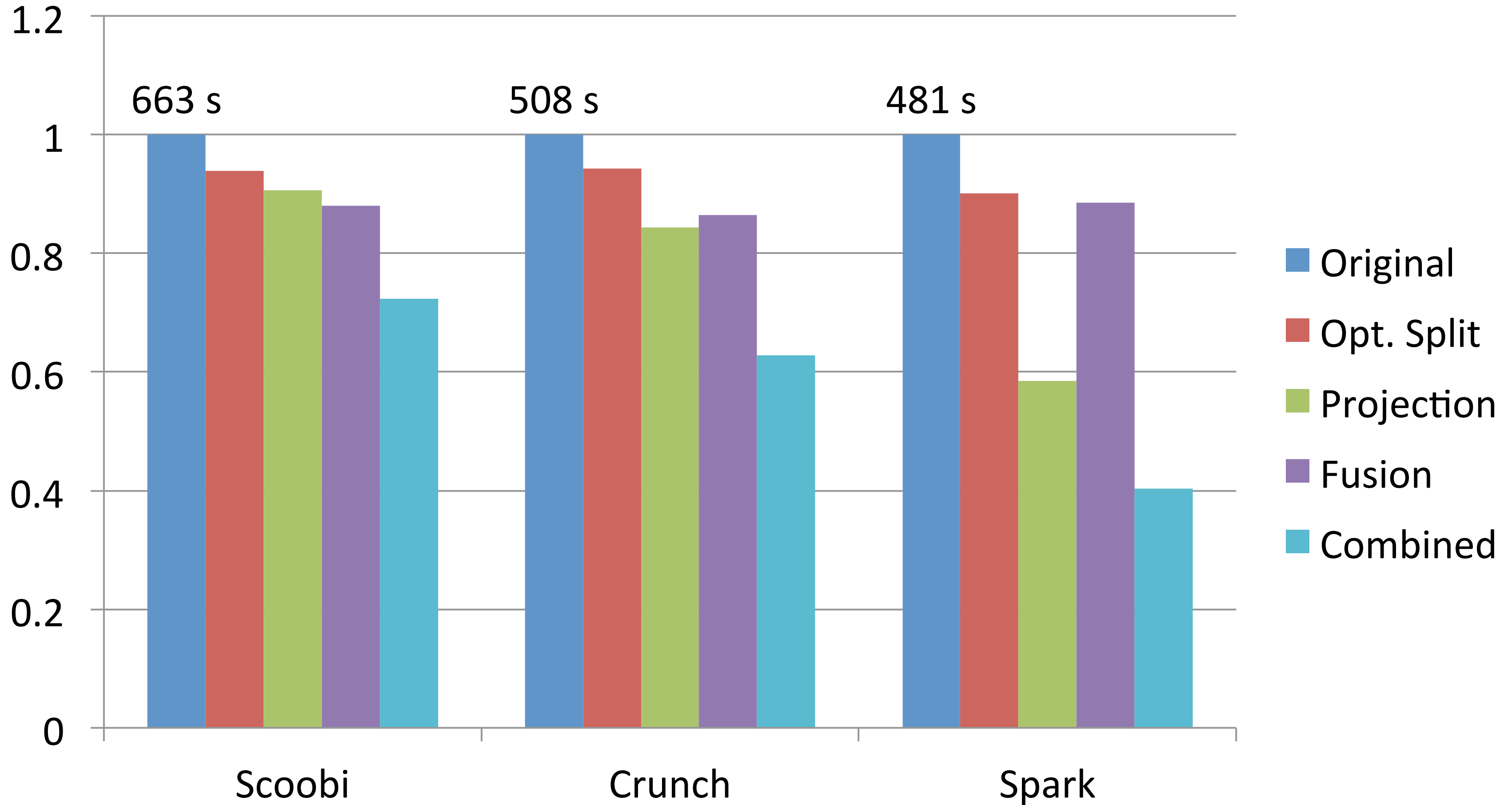
Results: Wordcount



Results: TPCH Q12

- TPCH Q12 reads from two collections, performs a join, and then reduces the output to two values (2 mapreduce jobs)
- Projection Insertion can remove most of the fields
- Input: dbgen with scaling factor 100 (~ 100Gb)

Results: TPCH Q12



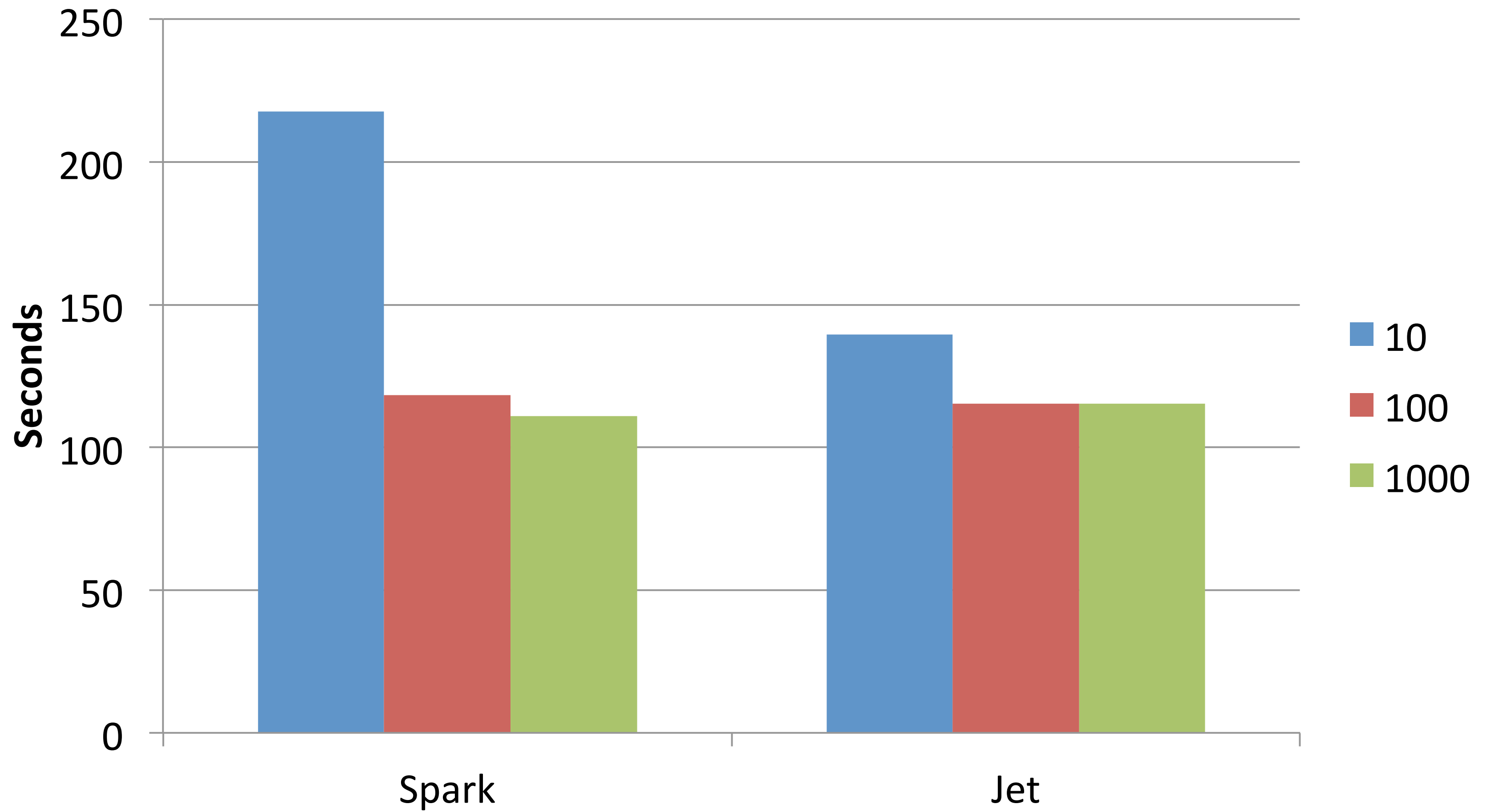
Results: KMeans

- KMeans is an iterative clustering algorithm
- Only tested in Spark, as it is 30x faster than Hadoop for this job
- Input data: 20 Gb, 50 Centers, 10 – 1000 dimensions

Results: KMeans

- Implementation taken from Spark repository
- Ported to Jet
- Extended Jet with an abstraction for multi-dimensional points, which generates arrays and while loops (no iterators)

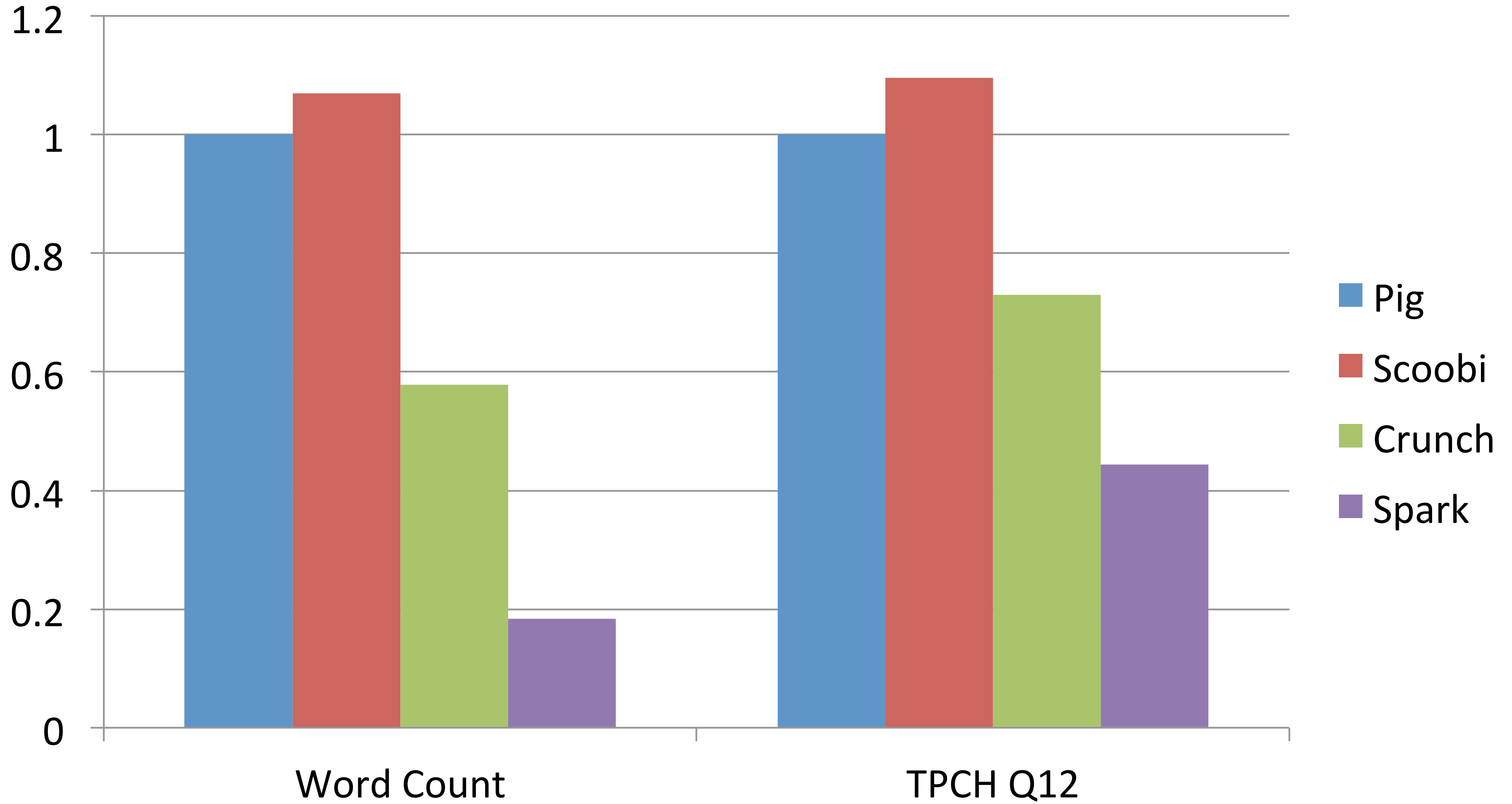
Results: KMeans



Jet vs Pig

- Pig's goals are similar to ours
- Optimizations are similar
  - Projection Insertion
  - Lazy parsing
- Pig only uses Hadoop

Jet vs Pig



Outline

- Background
- Optimizations
- Evaluation
- Conclusion

Future Work

- Add other optimizations
  - Relational optimizations (Reorder joins etc)
  - Move filters before joins
- Integrate with other LMS DSL's
  - Use GPU's
  - Regular Expressions

Projection Insertion

```
def project(in: Complex) = {
    Complex_0(in.re)
}
```

Fast

Concise

Portable

Extensible

Backup

- Parsing
  - How to define class, parsing method, etc
- Generated Writables
  - Bitset usage, switch
- Why not AoS to SoA