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

The overall MapReduce word count process

Input

Splitting

Mapping

Shuffling

Reducing

Final result

Deer Bear River
Car Car River
Deer Car Bear

Deer, 1
Bear, 1
River, 1

Car, 1
Car, 1
River, 1

Deer, 1
Car, 1
Bear, 1

Bear, 1
Bear, 1

Car, 1
Car, 1
Car, 1

Deer, 1
Deer, 1

River, 1
River, 1

Bear, 2

Car, 3

Deer, 2

River, 2

Bear, 2
Car, 3
Deer, 2
River, 2
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

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

Wordcount in Jet

User Defined Function in Jet

```scala
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

Object

Fields: Map[String, (Type, Value)]

Methods: Map[Signature, Method]
Background: Field Read Shortcut

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

becomes

```scala
val re = 1
```

No object required => No object created
Background: Decomposition

```scala
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(reOut, imOut)
  out: Complex
}
```

Constructor Invocation last
Outline

• Background
• Optimizations
  – Code Motion
  – Loop Fusion
  – Projection Insertion
• Evaluation
• Conclusion
Optimizations in MapReduce

The overall MapReduce word count process

Input

Splitting

Deer Bear River
Car Car River
Deer Car Bear

Mapping

Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1

Shuffling

Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1

Reducing

Bear, 2
Car, 2
Car, 3
Car, 1
River, 2
Deer, 2

Final result

Bear, 2
Car, 3
Deer, 2
River, 2

Reduce CPU Time

Reduce CPU Time
Optimizations: Code Motion

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

becomes

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

matches / split / replaceAll

RegexFrontend

create

Regex Pattern

Java Regex

Automaton

Fast Splitter
Optimizations: Loop Fusion

map (parse)

flatMap (parse, filter, tuple)

map (tuple)

String

LogEvent

LogEvent

(Long, LogEvent)
Optimizations in MapReduce

The overall MapReduce word count process

Input

Splitting

Mapping

Shuffling

Reducing

Final result

Deer Bear River

Car Car River

Deer Car Bear

Deer, 1
Bear, 1
River, 1

Car, 1
Car, 1
River, 1

Deer, 1
Car, 1
Bear, 1

Bear, 1
Bear, 1

Car, 1
Car, 1
Car, 1

Deer, 1
Bear, 1

Bear, 2

Car, 3

Deer, 2

River, 1

River, 1

River, 2

Bear, 2

Car, 3

Deer, 2

River, 2

Less Data
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

![Diagram of class hierarchy]

- **Complex**
  - `re`
  - `im`

- **Complex_0**
  - `re`

- **Complex_0_1**
  - `re`
  - `im`
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

```scala
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)
}
```

Loop Fusion
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)
}

Field Read Shortcut
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)
}
**Projection Insertion: Step 4**

```python
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: Analysis

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

How do we know which fields to keep?
Projection Insertion: Analysis

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

map (parse)

join

map

map (parse)

(join (Long, LogEvent), User)

(map Long, LogEvent)

(map Long, User)

(map Long, (LogEvent, User))
Projection Insertion: Propagation

```
String

(map (parse))

(Long, LogEvent)

(join)

(map)

{string}

(String)

(map (parse))

(Long, User)

(Long, (LogEvent, User))

(String)
```
Projection Insertion: Propagation

String

(Long, LogEvent)

map (parse)

{logevent.date, user.name}

join

(Long, (LogEvent, User))

map

{string}

String
Projection Insertion: Propagation

String

(Long, LogEvent) → map (parse) → join → map → {logevent.date, user.name} → (Long, (LogEvent, User)) → String

(Long, User)
Projection Insertion: Propagation

map (parse)

map (parse)

join

map

(String, {long, logevent.date})

{long, logevent.date} {long, user.name}

{logevent.date, user.name}

(Long, (LogEvent, User))

{string}

string

(Long, LogEvent)

(Long, User)
Projection Insertion: Propagation

map (parse)

(map parse Map)

map (parse)

(map parse Map)

join

(join Projector)

map

(map Map)

String

(Long, LogEvent)

{long, logevent.date}

{long, user.name}

(Long, (LogEvent, User))

{logevent.date, user.name}

(Long, User)

{string}

(String)
Projection Insertion: Fusion

- String
- (Long, LogEvent)
- map (parse)
- (Long, LogEvent)
- map (project)
- (Long, LogEvent)
- map (project)
- join
- (Long, LogEvent)
- {long, logevent.date}
- {long, logevent.date}
- {long, user.name}
- {long, user.name}
- join
- {logevent.date, user.name}
- {logevent.date, user.name}
- map
- {string}
- String
Optimizations: Mapper of TPCH Q12

String

map (parse)

filter

filter

(3 more filters)

map (tuple)

(Long, Lineltem)
Optimizations: Mapper of TPCH Q12

Unoptimized

```python
def x54(x51): (x54, x55) = (x56, x54)
    if x57 in {x58, x59}:
        return True
    if x54 in {x56, x57}:
        return False
```
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

- Scoobi: 913 s
- Crunch: 512 s
- Spark: 220 s

Bar chart showing performance across different tools and optimizations:

- Original
- 1) Fusion + Projection
- 2) 1 + Code Motion
- 3) 2 + Opt. Split
- 4) 3 + Opt. Automaton
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

<table>
<thead>
<tr>
<th></th>
<th>Scoobi</th>
<th>Crunch</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCH Q12</td>
<td>663 s</td>
<td>508 s</td>
<td>481 s</td>
</tr>
</tbody>
</table>

- Original
- Opt. Split
- Projection
- Fusion
- Combined
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

Seconds

Spark

Jet

- 10
- 100
- 1000
Jet vs Pig

• Pig’s goals are similar to ours
• Optimizations are similar
  – Projection Insertion
  – Lazy parsing
• Pig only uses Hadoop
Jet vs Pig

Word Count

TPCH Q12

Pig
Scoobi
Crunch
Spark
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

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

• Parsing
  – How to define class, parsing method, etc

• Generated Writables
  – Bitset usage, switch

• Why not AoS to SoA