Jet: An Embedded DSL for High Performance Big Data Processing

Stefan Ackermann
ETH Zürich
stefaack@student.ethz.ch

Vojin Jovanovic
EPFL
{first.last}@epfl.ch

Tiark Rompf
EPFL
{first.last}@epfl.ch

Martin Odersky
EPFL
{first.last}@epfl.ch

ABSTRACT

Cluster computing systems today impose a trade-off between generality, performance and productivity. Hadoop and Dryad force programmers to write low level programs that are tedious to compose but easy to optimize. Systems like Dryad/LINQ and Spark allow concise modeling of user programs but do not apply relational optimizations. Pig and Hive restrict the language to achieve relational optimizations, making complex programs hard to express without user extensions. However, these extensions are cumbersome to write and disallow program optimizations.

We present a distributed batch data processing framework called Jet. Jet uses deep language embedding in Scala, multi-stage programming and explicit side effect tracking to analyze the structure of user programs. The analysis is used to apply projection insertion, which eliminates unused data, as well as code motion and operation fusion to highly optimize the performance critical path of the program. The language embedding and a high-level interface allow Jet programs to be both expressive, resembling regular Scala code, and optimized. Modular design allows users to extend Jet with modules, which specify an abstract interface and how to generate high performance code for it. Through a modular code generation scheme, Jet can execute programs on both Spark and Hadoop. Compared with naïve implementations we achieve [**TODO: 149%**] speedups on Spark and [**TODO: 59%**] on Hadoop.

Keywords

Domain-specific Languages, Multi-stage Programming, MapReduce, Operation Fusion, Projection Insertion

1. INTRODUCTION

In the past decade, numerous systems for big data cluster computing have been studied [**?**, **?**, **?**, **?**, **?**]. Programming models of these systems impose a trade-off between generality, performance and productivity. Systems like Hadoop MapReduce [**?**], and Dryad [**?**] provide a low level general purpose programming model that allows writing fine grained and optimized code. However, low level optimizations greatly sacrifice productivity [**?**]. Models like Spark [**?**], FlumeJava [**?**] and Dryad/LINQ [**?**] provide high level operations and general purpose programming models, but their performance is limited by glue code between high level operations. Also, many relational optimizations are impossible due to the lack of knowledge about the program structure.

Domain-specific approaches, like Pig [**?**] and Hive [**?**], have a narrow, side effect free and domain-specific interface that allows them to be both productive and to apply relational optimizations. However, they come with their own set of limitations. Their programming model is often too simple for a wide range of problems. This requires reverting to user defined functions, which are cumbersome and again hard to optimize, or to abandoning the model completely.Moreover, there is the high overhead of learning a new language. Proper tool support, like debugging tools and IDE support is often limited. It is also hard to extend these frameworks with optimizations for a new domain.

Recently there have been several solutions that try to make programming distributed batch processing efficient, productive and general at the same time. Steno [**?**] implements an innovative runtime code generation scheme that eliminates iterators in Dryad/LINQ queries. It operates over flat and nested queries and produces a minimal number of loops without any iterator calls. Manimal [**?**] and HadoopToSQL [**?**] apply byte code analysis, to extract information about unused data columns and selection conditions; as a result to gain enough knowledge to apply common relational database optimizations for projection and selection. However, since these solutions use static byte code analysis they must be safely conservative which can lead to missed optimization opportunities.

This paper presents Jet a new domain-specific framework for distributed batch processing that provides a high level declarative interface similar to Dryad/LINQ and Spark. Jet builds upon language virtualization [**?**] and lightweight modular staging [**?**] (LMS) and has the same syntax and semantics as the regular Scala language with only a few restrictions. Because Jet includes common compiler and relational optimizations, as well as domain-specific ones, it produces efficient code closely comparable with hand optimized ver-

sions. It is designed in a composable and modular way - the code generation is completely independent of the parsing and optimizations - and allows extensions for supported operations, optimizations and back-ends.

This paper makes following contributions to the state of the art:

- We implement the Jet framework for distributed batch data processing that provides a general high level programming model with carefully chosen restrictions. These restrictions allow relational, domain-specific as well as compiler optimizations but do not sacrifice program generality.
- We introduce a novel projection insertion algorithm that operates across general program constructs like classes, conditionals, loops and user defined functions, takes the whole program into account and does not rely on safe assumptions which lead to missed optimization opportunities.
- We show that Jet allows easy language extension and code portability for distributed batch processing frameworks.

In Section **??** we provide background on LMS, language virtualization and big data frameworks. Then, in Section **??** we explain the programming model and present simple program examples. In Section **??** we explain the novel projection insertion optimization algorithm in Section **??** and Section **??** explains the fusion optimization. We evaluate Jet in **??** and discuss our approach in Section **??**. Jet is compared to state of the art in Section **??**, future work is in Section **??** and we conclude in Section **??**.

2. BACKGROUND

In this section we explain language virtualization in Scala, Lightweight Modular Staging (LMS) library [**?**, **?**], optimizations in LMS and distributed batch processing frameworks relevant for this paper.

2.1 Virtualized Scala

Jet is written in an experimental version of Scala called Virtualized Scala [**?**] which provides facilities for deep embedding of domain-specific languages (DSLs). Deep embedding is achieved by translating regular language constructs like conditionals, loops, variable declarations and pattern matching to regular method calls. For example, for the code `if (c) a else b`, the conditional is not executed but instead a method call is issued to the method `__ifThenElse(c, a, b)`. In case of deeply embedded DSLs this method overridden to create an intermediate representation (IR) node that represents the `if` statement.

In Virtualized Scala, all embedded DSLs are written within DSL scopes. These special scopes look like method invocations that take one by name parameter (block of Scala code). They get translated to the complete specification of DSL modules that are used in a form of a Scala trait mix-in composition[1]. For example: `stivoDSL{ \\ dsl code }` gets translated into: `new StivoDSL { def main(){...}}` This makes all the DSL functionality defined in StivoDSL visible in the body of the by name parameter passed to Jet method. Although modified, Virtualized Scala is fully binary compatible with Scala and can use all existing libraries.

[1]Scala's support for multiple inheritance

2.2 Lightweight Modular Staging

Jet is built upon the Lightweight Modular Staging (LMS) library. LMS utilizes facilities provided by Virtualized Scala to build a modular compiler infrastructure for developing staged DSLs. It represents types in a DSL with polymorphic abstract data type `Rep[T]`. A term inside a DSL scope that has a type `Rep[T]` declares that once the code is staged, optimized, and generated, the actual result of the term will have type `T`. Since `Rep[T]` is an abstract type, each DSL module can specify concrete operations on it, which are used for building the DSL's intermediate representation.

Since Scala's type system supports type inference and implicit conversions, most of the `Rep[T]` types are hidden from the DSL user. This makes the DSL code free of type information and makes the user almost unaware of the `Rep` types. The only situation where `Rep` types are visible is in parameters of methods and fields of defined classes. In our experience with writing DSLs, `Rep` types do not present a problem but gathering precise and unbiased information on this topic is very difficult.

The modular design of LMS allows the DSL developer to arbitrarily compose the interface, optimizations and code generation of the DSL. Module inclusion is simply done by mixing Scala traits together. The correctness of the composition and missing dependencies are checked by the type system. Code generation for a DSL is also modular, so the effort is almost completely spent on domain-specific aspects. LMS provides implementations for most of the Scala constructs and the most common libraries. This allows to make DSLs that are fairly general - comparable to standard Scala.

Unlike Scala, which does not have an effect tracking mechanism, LMS provides precise information about the effect for each available operation. The DSL developer needs to explicitly specify the effects for each DSL operation he introduces. The LMS effect tracking system then calculates the effects summary for each basic block in the DSL code. This allows the optimizer to apply code motion on the pure (side effect free) parts of the code. All implementations for standard library constructs that LMS provides such as strings, arrays, loops and conditionals, already include effect tracking.

LMS builds a complete intermediate representation of the code, optimizes it, and then generates optimized code for the chosen target language. The generated code has then to be compiled itself, and then it can be invoked. If the compilation delay is deemed inappropriate for a certain use case, it is possible to execute the code directly in Scala. A shallow DSL embedding can be achieved this way, instead of building the IR.

In Listing **??**, we show a simplified version of a reusable DSL module for printing. In trait `PrintlnExp` we define how the `println` operation is linked to the intermediate

```
def parse(st: Rep[String]) = {
  val sp = st.split("\\s")
  Complex(Float(sp(0)), Float(sp(1)))
}
val x = new Array[Complex](input.size)
for (i <- 0 to input.size) {
  x(i) = parse(input(i))
}
for (i <- 0 to x.size) {
  if (x(i).im == 0) println(x(i).re)
}
```

(a) Original program

```
val size = input.size
val x = new Array[Complex](size)
for (i <- 0 to size) {
  val pattern = new Pattern("\\s")
  val sp = pattern.split(input(i))
  x(i) = Complex(Float(sp(0)), Float(sp(1)))
}
for (i <- 0 to x.size) {
  if (x(i).im == 0) println(x(i).re)
}
```

(b) CSE and inlining

```
val size = input.size
val re = new Array[Float](size)
val im = new Array[Float](size)
for (i <- 0 to size) {
  val pattern = new Pattern("\\s")
  val sp = pattern.split(input(i))
  re(i)  =  Float(sp(0))
  im(i) = Float(sp(1))
}
for (i <- 0 to size) {
  if (x(i).im == 0) println(x(i).re)
}
```

(c) AoS → SoA

```
val size = input.size
val pattern = new Pattern("\\s")

for (i <- 0 to size) {
  val sp = pattern.split(input(i))
  val im = Float(sp(1))
  if (im == 0)  {
    val re = Float(sp(2))
    println(re)
  }
}
```

(d) Loop fusion and code motion

Figure 1: Step by step optimizations in LMS

```
// creates the println statement in the IR
trait PrintlnExp extends BaseExp {
  def println[T](st: Rep[String]) =
    reflectEffect(PrintlnNode(st))
}
trait PrintlnGen extends ScalaGen {
   def emit(node: Rep[Any]) = node match {
       case PrintlnNode(str) =>
          println("println("+str+")")
   }
}
```

Listing 1: Example of how the DSL module is specified. This module is used for measuring a performance of a block of code and can be reused in any other Scala backed DSL.

representation. The `reflectEffect` method defines that the `profile` method has global side effects which signals the compiler that it can not be reordered with respect to other globally effectful statements or be moved across control structures. In the `PrintlnGen` trait, we define how the code for `println` is generated for Scala.

LMS has been used successfully by Brown et al. for heterogeneous parallel computing in project Delite [**?**, **?**] and by Kossakowski et al. for a JavaScript DSL [**?**].

2.3 LMS Optimizations

When writing DSLs, the DSL author can exploit their domain knowledge to apply high level optimizations and program transformations. Afterwards, the program is usually lowered to a representation closer to the actual generated code. LMS provides a set of common optimizations for the lowered code, which are: common subexpression elimination (CSE), dead code elimination (DCE), constant folding (CF) and function inlining. LMS also applies code motion, which can either: *i)* move independent and side effect free blocks out of hot loops *ii)* move code segments that are used inside conditionals but defined outside, closer to their use site.

Another interesting optimization is the transformation of an array of structural types to a structure of arrays (AoS → SoA), each containing only primitive fields. This transformation removes unnecessary constructor invocations and enables DCE to collect unused fields of an structure. It can be applied to built-in data structures like tuples as well as immutable user-defined types. It is similar in effect to row storage in databases and it gives great performance and memory footprint improvements.

LMS also provides a very general mechanism for operation fusion that uses standard loops as the basic abstraction. It is better than existing deforestation approaches since it generalizes to loops and can apply both vertical and horizontal fusion. In vertical fusion, the algorithm searches for producer consumer dependencies among loops, and then fuses their bodies together. In horizontal fusion, independent loops of the same shapes are fused together and index variables are relinked to the fused loop's index variable. Fusion greatly improves performance as it removes intermediate data structures and uncovers new opportunities for other optimizations.

In Listing **??**, we present these optimizations on a single example which parses an array of complex numbers and prints only the real parts of them. Step **??**) shows the original program, **??**) shows how CSE extracts `size` and inlining replaces `parse` and `split` invocations with their bodies. In step **??**) the array `x` of complex numbers is split into two arrays of floating points. In **??**) the loops are fused together, which then allows code motion to move the constant pattern out of the loop and move the parsing of the real component

into the conditional. The intermediate arrays can then be removed by DCE.

2.4 Distributed Batch Processing Frameworks

Jet generates Scala code for Crunch [**?**], Scoobi [**?**] and Spark [**?**]. Both Crunch and Scoobi use Hadoop as the execution engine and provide an MSCR implementation as presented in [**?**]. Crunch is implemented in Java and provides a rather low level interface, in which the user must provide implementation for user classes. Scoobi on the other hand is a Scala framework, which features a declarative high level interface and creates efficient serialization for user classes with only a minimal amount of help required.

Spark is a recent execution engine which makes better use of the cluster's memory, explicitly allowing the user to cache data. This allows huge speedups on iterative jobs which can reuse the same data multiple times, unlike with Hadoop. It also features a declarative high level interface and has support for multiple serialization frameworks and it also features a shell for low latency interactive data querying.

3. PROGRAMMING MODEL

The basic abstraction in our programming model is the interface `DColl[T]`. `DColl[T]` represents a distributed collection of elements that have type `S` which is a subtype of `T`. The elements of a `DColl[T]` collection are immutable, so each operation on the list can only: *i)* produce a new `DColl`, *ii)* save it to persistent storage, *iii)* materialize it on the master node or *iv)* return an aggregate value.

`DColl` operations are presented in Table **??**. In the left column, we show which frameworks support which method. The middle column shows the method name. Finally, the right column contains the type of `DColl` that the operation is called on, and return type of the operation.

Operations `DColl()` and `save` are used for loading and storing data to the persistent storage. `map, filter` and `flatMap` are standard list comprehensions for transforming the data by applying the argument function and can also be used with Scala `for` comprehensions. Operations `groupByKey`, `join`, `cogroup`, `cross` and `reduce` are applicable only if the elements of `DColl` form a key/value tuple. `reduce` is used for general aggregation after the `groupByKey`, `join, cogroup` and `cross` are different types of relational joins. `sort` sorts the dataset, `partitionBy` defines partitioning among machines, and `cache` signals that data should be kept in cluster memory for faster future accesses. Two `DColl`s can be concatenated by the `++` operation. A DColl can be materialized on the master node by calling `materialize()`.

Some methods accept functions as their parameters. Code within these functions can be either written in the Jet DSL, or by using existing functions from an external library or common JVM libraries. Using JVM libraries requires just one extra line of code per method.

In Listing **??**, we show an implementation of a simple word count example, in which the code does not have any visible `Rep` types. Since a large subset of the Scala library is implemented as a DSL module, functions like `split` and string

```
val read = DColl("hdfs://..." + input)
val parsed = read.map(WikiArticle.parse(_))
parsed.flatMap(_.split("\\s"))
  .map(x => (x, 1))
  .groupByKey()
  .reduce(_ + _)
  .save("hdfs://..." + output)
```

Listing 2: Example of word count program where type inference removes the need to declare any `Rep` types.

concatenation are used the same way as they are in Scala. In the second line, the regular (with arguments wrapped in `Rep`) method `parse` is passed to the `map` method. Pig and Hive do not have functions in their own language, but allow writing user defined functions in other languages which requires a considerable amount of boilerplate code.

All methods except for `cache` and `sort` can be mapped to methods in Scoobi, Spark and Crunch. Other back-ends (including Dryad) provide these primitives as well. The `cache` method currently works with Spark only but it can be added to the interface of other back-ends, in which it would have no effect, such that the code stays portable. From existing frameworks today only HaLoop [**?**] and Twister [**?**] can benefit from it, however we did not implement code generation for them. Method `sort` is inconsistent in most of the frameworks so we have not mapped uniformly to all of them. However, with slight modifications to the framework implementations it could be supported as well. `sort` can also be implemented in Jet itself by using `takeSample` and `partitionBy`.

4. OPTIMIZATIONS

In this section we present the projection insertion and the operation fusion optimizations implemented in Jet.

4.1 Projection Insertion

A common optimization in data processing is to remove intermediate values early that are not needed in later phases of the computation. It has been implemented in relational databases for a long time, and has recently been added to the Pig framework. This optimization requires all field accesses in the program to be explicit. A library can provide this, but its usage is more intrusive than if the framework can use compiler support.

In Jet, we support this optimization for algebraic data types, more specifically final immutable Scala classes with a finite level of nesting. Our approach does not require special syntax or access operators and supports method declarations on data types just like methods of regular Scala classes. While implementing our benchmarks we found this to be a reasonably expressive model for big data programming. The DSL user needs to supply class declarations, from which we generate all the necessary code for its use in Jet.

A projection insertion optimization needs to know about the liveness of all fields it can possibly remove. AoS → SoA optimization in LMS provides this within a scope, by decomposing all control structures to multiple copies of it accessing each field separately. After DCE, all remaining fields are alive within that scope. For Jet, this is not enough, as we

Framework	Operation	Transformation
All	`DColl(uri: Rep[String])`	`String => DColl[T]`
	`save(uri: Rep[String])`	`DColl[T] => Unit`
	`map(f: Rep[T] => Rep[U])`	`DColl[T] => DColl[U]`
	`filter(f: Rep[T] => Rep[Boolean])`	`DColl[T] => DColl[T]`
	`flatMap(f: Rep[T] => Rep[Iter[U]])`	`DColl[T] => DColl[U]`
	`groupByKey()`	`DColl[(K, V)] => DColl[(K, Iter[V])]`
	`reduce(f: (Rep[V], Rep[V]) => Rep[V])`	`DColl[(K, Iter[V])] => DColl[(K, V)]`
	`cogroup(right: Rep[DColl[(K, W)]])`	`DColl[(K, V)] => DColl[(K, (Iter[K], Iter[W]))]`
	`join(right: Rep[DColl[(K, W)]])`	`DColl[(K, V)] => DColl[(K, (V, W))]`
	`++(other: Rep[DColl[T]])`	`DColl[T] => DColl[T]]`
	`partitionBy(p: Rep[Partitioner[T]])`	`DColl[T] => DColl[T]`
	`takeSample(p: Rep[Double])`	`DColl[T] => Iter[T]`
	`materialize()`	`DColl[T] => Iter[T]`
Spark	`cache()`	`DColl[T] => DColl[T]`
	`sort(cmp: Rep[Comparator[T]])`	`DColl[T] => DColl[T]`
Crunch	`sort(asc: Rep[Boolean])`	`DColl[T] => DColl[T]`

Table 1: DColl operations and their framework support. For clarity reasons, `Iter` represents the Scala `Iterable` and `Rep[_]` types in the rightmost column are omitted.

support operations in the data-flow graph, like `groupByKey`, for which we found no good way to decompose. However, we can define a liveness analysis for each operation in our programming model. For all operations in our programming model we identify rules on how this operation influences the liveness of fields. For operations that have a closure parameter, we apply AoS → SoA inside that closure to acquire liveness information.

By performing this analysis on each node in reverse topological order and propagating the liveness information to its predecessors, we are able to perform removal of unused fields in all operations. On a distributed program, the removal of dead fields is especially important before an operation that requires network transport of an object or stores it in memory. We call such an operation a barrier, and insert a projection which only contains the live fields before it.

Since we support nested classes of a finite level, the nested fields of a class form a tree, and if a field in such a tree is alive, it requires liveness of all its ancestors. We call the path of a nested field to the root of the tree an *access path*, and represent it using a string. The Figure **??** shows the tree of nested fields for the class Tuple2[String, A]. The nodes describe the class of a parent's field, while the edges represent the field name. The *access path* to each nested field is formed by concatenating the edges with a separating dot. In the Figure, the access path for the field `id` in class `B` would be `_2.b.id`. For each edge in the data-flow graph, that our operations form, we need to compute the set of access paths.

```
case class A(id: String, b: B)
case class B(id: String)
val t = ("tuple", A("a", B("b")))
t: scala.Tuple2[String, A]
```

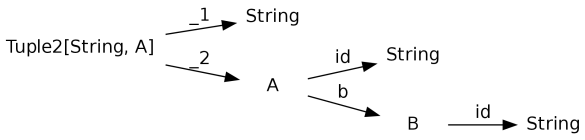

Figure 2: Visualization of the tree of fields for class Tuple2[String, A]

For each operation, we need to define how the access paths used by it are translated into the access paths it uses. For this analysis we used following primitives:

- *Access paths for a type*: Given a type, this primitive creates access paths for all the nested fields within it. In Figure **??** a `DColl.save()` with elements of class A returns the access paths $id, b, b.id$.
- *Closure analysis*: This primitive returns a list of all access paths on the closure's input type.
- *Rewrite access paths*: Several operations have semantics which influence the type and therefore the access paths. For example, the `cache` operation will always have the same input and output type and is known not to change any fields, so all access paths from its successors must be propagated to its predecessors. The `groupByKey` operation on the other hand always reads all nested fields of the key, and has to rewrite all access paths of the value.
- *Narrow closure*: Given a closure, this primitive replaces the closure's original output with a narrowed one based on the access paths of its successors.

To analyze a `map` operation, we need to combine the narrow closure and the closure analysis primitive. AoS → SoA ensures that the output symbol of a closure is always a constructor invocation. We apply the narrow closure primitive to create a new closure, in which the new output reads from the old output only the live fields. LMS recognizes when a field is read from a constructor invocation in the same scope, and instead of generating an access for a field, it returns that value directly. This happens for all the fields, therefore the old constructor invocation will not be read anymore, and DCE will remove it. This means that the field values, which only it was reading, will also not be read anymore, and they too will be eliminated. When the closure has been cleaned of dead values, we can analyze this new closure to get the access paths from it.

Table **??** shows how relevant operations in Jet are analysed and how access paths are propagated. The first column is

Operation	Access Path Computation	Barrier
`filter`	$P = S + analyze(f)$	
`map`	$P = analyze(narrow(f))$	
`groupByKey`	$P = all(I, \_1) + rewrite(S, \_2.iterable.x \Rightarrow \_2.x)$	✓
`join`	$P_L = all(I, \_1) + rewrite(S, \_2.\_1.x \Rightarrow \_2.x)$	✓
	$P_R = all(I, \_1) + rewrite(S, \_2.\_2.x \Rightarrow \_2.x)$	
`reduce`	$P = rewrite(analyze(f), x \Rightarrow \_2.iterable.x) + rewrite(S, \_2.x \Rightarrow \_2.iterable.x)$	
`cache`	$P = S$	✓
`save`	$P = all(I)$	

Table 2: Access path computation and propagation for selected operations. P and S are the access path sets of the predecessor and successor. $analyze(f)$ analyzes the given closure f, $narrow(f)$ narrows it. $rewrite(S, x.y \Rightarrow x.z)$ rewrites the access paths in S with prefix $x.y$ to have prefix $x.z$ instead. I is the input element type of the operation, and all(I) can be used to generate the input types.

the operation name, the middle column contains the rules for propagating the access paths and the last column shows whether we treat this operation as a barrier.

4.2 Operation Fusion

In Section **??** we have shown that the `DColl` provides declarative higher order operations. Closures passed to these operations do not share the same scope of variables. This reduces the number of opportunities for the optimizations described in Section **??**. Moreover, each data record needs to be read, passed to and returned from the closure. In both the push and the pull data flow model this enforces expensive virtual method calls [**?**] for each data record. To reduce the performance penalty we have implemented fusion of monadic operations `map`, `flatMap` and `filter` through the underlying loop fusion algorithm described in Section **??**.

The loop fusion optimization described in Section **??** supports horizontal and vertical fusion of loops as well as fusion of nested loops. Also, it provides a very simple interface to the DSL developer for specifying loop dependencies and for writing fusible loops. We decided to extend the existing mechanism to the `DColl` operations although they are not strictly loops. We could have taken the path of Murray et al. in project Steno [**?**] by generating an intermediate language which can be used for simple fusion and code generation. Also, we could use the Coutts et al. [**?**] approach of converting `DColl` to streams and applying equational transformation to remove intermediate results. After implementing the algorithm by reusing loop fusion we are confident that it required significantly less effort than reimplementing existing approaches.

Before the fusion optimization, the program IR represents an almost one to one mapping to the operations in the programming model. Each monadic operation is represented by the corresponding IR node which carries its data and control dependencies and has one predecessor and one or more successors. On these IR nodes we first apply a lowering transformation, which translates monadic operations to their equivalent loop based representation. Described transformation is achieved by the program translation described in Listing **??**. These rules introduce two new IR nodes: *i)* $shape\_dep(n, m)$ that carries the explicit information about its vertical predecessor n and a $prevent\_fusion$ bit m, and *ii)* $iterator\_value$ that represents reading from an iterator of the preceding `DColl`. *shape_dep* replaces the shape variable (e.g. `in.size`) of the loop IR node. The *yield* operation represents storing to the successor collection and is used in the LMS fusion algorithm for correct merging of two loops.

We define the set of program `DColl` nodes as D.
For $n \in D$:
  $pred(n)$ `gives the predecessor of the node in` D
  $succ(n)$ `returns a set of node successors in` D
  $prevent\_fusion(n) = |succ(pred(n))| > 1$
Lowering Transformations:
  $out = map(in, op) \rightarrow$
    $loop(shape\_dep(in, prevent\_fusion(in)), index, \{$
      $yield(out, op(iterator\_value(in))$
    $\})$
  $out = filter(map, op) \rightarrow$
    $loop(shape\_dep(in, prevent\_fusion(in)), index, \{$
      $if(op(iterator\_value(in))$
        $yield(out, iterator\_value(in))$
    $\})$
  $out = flatMap(in, op) \rightarrow$
    $loop(shape\_dep(in, prevent\_fusion(in)), index, \{$
      $w = op(iterator\_value(in))$
      $loop(w.size, index, \{yield(out, w(index))\})$
    $\})$

Figure 3: Operation lowering transformations.

For the back-end frameworks that Jet supports, the fusion operation is not possible for all operations in the data flow graph. If one node has multiple successors, after fusion, it would form a loop that yields values to multiple `DColl`s. This would be possible for Hadoop, as it supports multiple outputs in the `Map` phase but is not currently supported in Spark, Scoobi and Crunch. To prevent fusion of such loops we added the $prevent\_fusion$ bit to the *shape_dep* node. We also prevent horizontal fusion by making *shape_dep* nodes always different in comparison.

After the lowering transformation we apply the loop fusion optimization from LMS. It iteratively fuses pairs of loops for which the consumer does not have the $prevent\_fusion$ bit set until a fixed point is reached. In each fusion iteration all other LMS optimizations are applied as well. Afterwards, in the code generation layer for each back-end, we specify how to compile loops with *shape_dep* nodes to the most general operation (e.g. the Hadoop `Mapper` class) that the framework provides. With this approach we could also generate code directly for Hadoop MapReduce which would result in a single highly optimized loop per `Mapper`. However, after doing experiments we concluded that the gain is not significant compared to using higher level back-ends. Therefore, as an alternative, we used the frameworks Scoobi and Crunch.

Unlike MapReduce based back-ends, Spark's design uses the pull data-flow model, implemented through iterators. Generating code for the pull data-flow model from the loop based

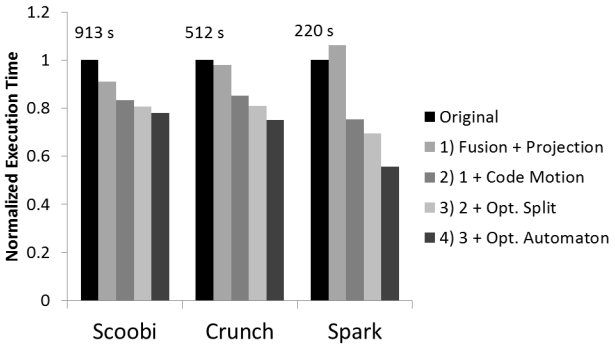

Figure 4: Word Count benchmark.

model (push data-flow) proved to be non-trivial. After evaluating different types of queues and array buffers we have decided to buffer intermediate results in a 4 MB array.

5. EVALUATION

We evaluate Jet optimizations by comparing the performance of three programs: *i)* A word count with prior text parsing, *ii)* TPCH [**?**] query 12 and a *iii)* k-means application. To evaluate the extensibility of the framework we introduce a new DSL component that represents vectors in the k-means benchmark.

All experiments were performed on the Amazon EC2 Cloud, using 20 "m1.large" nodes as slaves and one as a master. They each have 7.5 GB of memory, 2 virtual cores with 2 EC2 compute units each, 850 GB of instance storage distributed over 2 physical hard drives and have 1 Gb/s network interface. Prior to the experiments we have measured up to 50 MB/s between two nodes. For the Hadoop experiments we used the cdh3u4 Cloudera Hadoop distribution. On top of it we used Crunch version 0.2.4 and Scoobi 0.4.0. We did not tweak Hadoop configuration beyond the default settings set by the Whirr 0.7.1 [**?**] tool. For benchmarking Spark we used the Mesos [**?**] EC2 script to start a cluster, and the most recent version of Spark for our tests. For Spark we changed the default parallelism level to the number of cores in the cluster and increased the maximum memory to 6GB.

While doing preliminary benchmarking we found some easy tweaks focused on regular expressions that we needed to include in Jet in order to have a fair comparison against Pig, which contains them. We implemented a fast splitter, which uses an efficient character comparison whenever the regular expression allows this. Additionally we select based on the regular expression between Java's implementation and the dk.brics.automaton library [**?**].

For serialization of data we used LMS code generation to achieve minimal overhead for both Crunch and Scoobi frameworks because they outperformed the Kryo [**?**] library by a small margin. For Spark we used the standard serialization mode which uses Kryo. All benchmarks were run three times and in the figures we present the average value. We also computed the standard deviations but we omitted them since they are smaller than 3% in all the experiments.

We made the Jet code, as well as the generated code, available on `https://github.com/jet-framework/jet`.

5.1 Parsing and Word Count

In this benchmark we evaluate the performance of Jet compiler optimizations without focusing on projection insertion. We choose a word count application that, prior to the inexpensive network shuffle, parses the input with 5 regular expressions making this job CPU bound. For this evaluation we start with an the original version of the program and add optimizations one by one. We first add the operation fusion and projection insertion optimizations. We then include code motion that removes regular expression compilation out of hot loops. Next we add the fast splitter and for the fully optimized version we use the optimized automaton regular expression library.

Our input is a 62 GB set of plain text version of Freebase Wikipedia articles. Our regular expressions are used clean up articles from strings that are not text words. This benchmark does not benefit from projection insertion but we include it for the comparison with the Pig framework in Section **??**.

In Figure **??** we show the job times for these versions normalized to the original program version. Performance improvements of all optimizations combined are from 29% for Scoobi, 33% for Crunch and 79% for Spark. The base performance of the frameworks differ by a large margin for this benchmark. Scoobi profits the most in this case from the fusion which indicates that the framework imposes additional overhead for declarative operations. In Spark, we notice larger benefits from our optimizations. We argue that it has significantly smaller IO overhead so that the optimizations have a bigger impact. Also, we notice that fusion optimization with projection insertion is slower than the original program for Spark. This result does not match our experiments in a smaller cluster setup, we believe that it could be caused by a straggler node in the cloud environment.

5.2 TPCH Query 12

This benchmark evaluates all optimizations combined but emphasizes the projection insertion. We chose the TPCH query 12 which includes an expensive `join` operation after which only two columns of the original data are used, thus giving projection insertion opportunity to eliminate unused columns. We compare the original program to each optimizations separately and all of them combined. As the data set we use a 100 GB plain text input generated by the DbGen tool [**?**].

In Figure **??** we show job times for different optimizations normalized to the original program version on different frameworks. We notice that projection insertion gives 20% percent better performance on Crunch and 11% on Scoobi. On Spark, the projection insertion improves the performance by 40%, significantly more than for the Hadoop based frameworks. We believe the difference is caused by Hadoop spills of the data to disk earlier, while Spark tries to keep it in the memory before it spills it. Spark is therefore very sensitive to memory overhead of the large `join`.

In this benchmark the optimizations interact with each other. The absolute performance gain for combined optimizations is 3% greater for Crunch, equal for Scoobi and 9% smaller for Spark than the sum of the absolute individual gains.

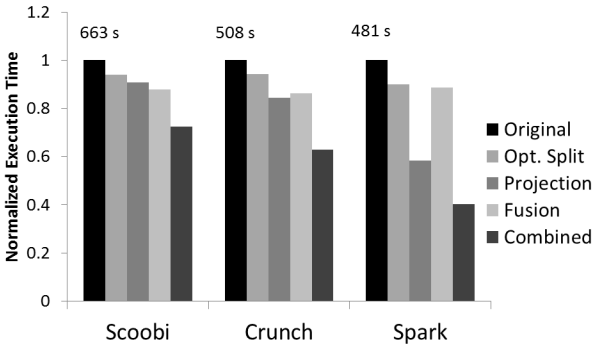

Figure 5: TPCH query 12 benchmark.

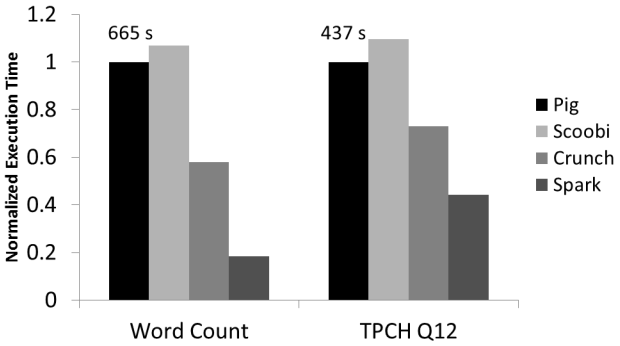

Figure 6: Comparison between Pig, Scoobi, Spark and Crunch.

5.3 Comparison with Pig

In Figure **??** we compare the most optimal versions of benchmarks to equivalent Pig programs. The y axis is normalized to the Pig execution time and overall job time is stated above the bar. We notice that for TPCH query 12 the combination of fusion, code motion and field reduction outperform Pig when the Crunch framework is used.

For the sake of showing comparison between the Hadoop based frameworks and the Spark framework we include the Spark results in the graph. We see that in all cases except for unoptimized TPCH query 12 it significantly outperforms the Hadoop based frameworks.

In the word count benchmark Crunch outperforms Pig even without any optimizations applied. We believe that Pig uses inefficiency string processing primitives. With all optimizations Crunch is 73% faster than Pig. We explain this by the more optimal regular expressions processing support included in the Jet. In regular expressions used in the benchmark Pig falls back to default Java regular expressions while Jet uses optimized automaton library. Scoobi framework performs slower than Pig in both benchmarks even with all optimizations applied.

5.4 Extensibility and Modularity

To evaluate modularity and extensibility of Jet we decided to extend with a `Vector` abstraction that has abstract methods for arithmetic vector operations that get compiled into loops over arrays. We also chose this benchmark to emphasize the extensibility of Jet.

We took a version of Spark k-means program [**?**] application and ported it to Jet. This application can neither benefits

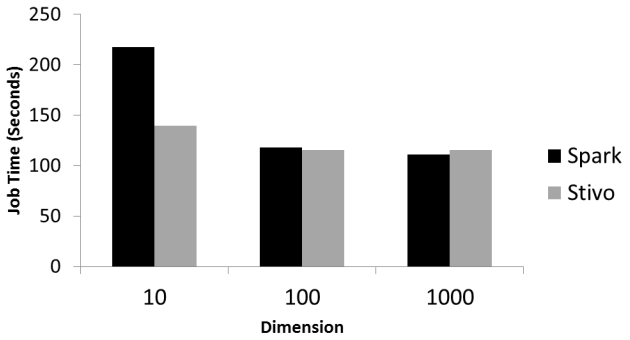

Figure 7: K-means benchmark.

from projection insertion reduction nor from operation fusion. We extended our DSL for this program with a highly optimized vector type that has all its operations iterating over the dimensions compiled into while loops. We only evaluate this benchmark on Spark, since it uses operations only defined in Spark and since it is known to outperform Hadoop by a large margin. As input we use synthetic data with 10 to 1000 dimensions, 100 centers and we keep the $dimensions * points$ factor constant so that each input file is around 20Gb.

Our results are similar to those described by Murray et al. in [**?**]. In lower dimensions our optimization shows large speedup while for 1000 dimensions our version performs slightly worse. We believe that the iterator overhead is quite high in case of 10 dimensions, such that our loops which removes it performs much better. At higher dimensions it's possible that the JVM can do a better job optimizing if the code is smaller, such that our pre-optimized and larger code becomes slightly slower. In any case our implementation seems favorable as it performs more consistently for different dimensions.

The `Vector` class extension is only 100 lines of Scala code and was implemented in one working day which shows that it is not hard to add highly optimal abstraction. Also, during evaluation we not getting satisfying performance in the Scoobi framework so we implemented the code generation for Crunch. The code for this module is 300 lines of code and the implementation took 4 man days.

6. DISCUSSION

The language we provide is the same as Scala in its basic constructs, however it does not support all of the functionalities. The following functionalities are not available:

- Projection optimization can only be applied to final immutable classes. This somewhat limits the language, but in large big data processing, data records are often not polymorphic.
- Polymorphism is supported only in a limited form. The limitation is that all possible implementations need to be known at staging time and currently it prevents optimization of the polymorphic method call.
- The whole Scala library is not available in its DSL form. This however does not limit the user since JVM methods can be used in the native form, but they can

not be optimized. Due to language embedding, for using JVM methods there is no boilerplate code.

One of the caveats of the staged DSL approach is that the program staging, compilation, generation and compilation of the generated code increases the startup time for the task. For the benchmarks we have evaluated that this process takes from 4 to 14 seconds. Although this can be significant, it needs to be only done once on a single machine so we believe it is not a limiting factor for batch jobs.

The only case where compile time becomes relevant is with back-ends that support interactive data analytics, like the Spark framework. Spending more than a couple of seconds for compilation would affect the interactivity.

However, if data processing is comparable to user reaction time, optimizations are not needed and we could implement a version of Jet which does not generate the IR, but executes the original code straight away. This feature is not implemented in Jet, but Kossakowski et al. [**?**] have done this for the Javascript DSL.

Each job requires a framework specific configuration for its optimal execution (e.g. the number of mappers, reducers, buffer sizes etc.). Our current API does not include tuning of these parameters, but in the future work we want to introduce a configuration part of the DSL to unify configuration of different backends. With the current programming model it is not possible to tune these parameters in a unified way.

7. RELATED WORK

Pig [**?**] is a framework for writing jobs for the Hadoop platform using an imperative domain-specific language called Pig latin. Pig latin's restricted interface allows the Pig system to apply relational optimizations that include operator rewrites, early projection and early filtering to achieve good performance. It has extensive support for relational operations and allows the user to choose between different join implementations with varying performance characteristics. Pig latin users need to learn a new language which is not the case with frameworks such as Hadoop, Hive and Crunch. It does not include user defined functions, the user needs to define them externally in another language, which will often prevent optimizations as Pig can not analyze those. Pig latin is not Turing complete as it does not include control structures itself. The language is not statically type checked so runtime failures are common and time consuming. Also, pure Java approaches benefit from a rich ecosystem of productivity tools.

Even though Jet also adopts a domain-specific approach, it is deeply embedded in Scala, Turing complete and allows the user to easily define functions which do not disable the optimizations. Currently Jet does not have support for many relational optimizations. However, it includes compiler optimizations and it is well extensible.

Steno [**?**] is a .NET library that, through runtime code generation, effectively removes abstraction overhead of the LINQ programming model. It removes all iterator calls inside LINQ queries and provides significant performance gains in CPU intensive jobs on Dryad/LINQ. Queries that use Steno do not limit the generality of the programming model but optimizations like code motion and early projection are not possible. Jet also removes excess iterator calls from the code but during operation fusion it enables other optimizations, especially when combined with early projection. The drawback of Jet is that it is slightly limited in generality.

Manimal [**?**] and HadoopToSQL [**?**] perform static byte code analysis on Hadoop jobs to infer different program properties that can be mapped to relational optimizations. They both use the inferred program properties to build indexes and achieve much more efficient data access patterns. Manimal can additionally organize the data into columnar storage. These approaches are limited by the incomplete program knowledge which is lost by compilation and runtime determined functions. They both do not restrict the programming model at all. Jet shares the idea of providing code generality to these approaches. However, it currently does not include data indexing schemes which could enable big performance improvements. We believe that the full type and program information available in Jet will enable us to build better data indexing schemes for a larger set of user programs.

8. FUTURE WORK

From the wide range of relational optimizations, Jet currently supports only early projection. In the future work we plan to introduce early filtering which will push filter operations before the expensive operators that require a barrier. Also, we plan to include program analysis phase which will allow building of indexes.

Text and XML processing are often processed in cluster computing and efficient processing of it can greatly reduce cost and energy consumption. With that in mind we plan to integrate Jet with other text parsing DSLs that are deeply embedded into the standard library. If prototyping shows that performance gains are significant we will add DSL modules for regular expressions, Scala parser combinators and XML library.

Jet currently only operates on distributed datasets so programs written in it can not be used for in memory data. We plan to integrate Jet with Delite [**?**] collections DSL which supports very efficient execution of batch operations. Delite also allows running queries on heterogeneous hardware architectures where jobs are scheduled for execution on both CPU and GPU processors.

9. CONCLUSION

We have presented the big data processing library Jet that provides an expressive high level programming model. Jet uses language virtualization, lightweight modular staging and side effect tracking to analyze user programs at runtime. This allows Jet to apply projection insertion, code motion as well as operation fusion optimizations to achieve high performance for declarative programs. Through modular code generation Jet allows execution on Spark, Crunch and Scoobi. Presented optimizations result in speedups of 148% in Spark, 59% in Crunch and 38% in Scoobi.

Unlike existing domain-specific approaches Jet provides high performance, general and expressive programming model which is integrated into the Scala language. It allows high performance user extensions and code portability between different big data processing frameworks.