Jet: An Embedded DSL for High Performance Big Data Processing

Stefan Ackermann
ETH Zürich
stefaack@student.ethz.ch

Vojin Jovanović
EPFL, Switzerland
vojin.jovanovic@epfl.ch

Tiark Rompf
EPFL, Switzerland
tiark.rompf@epfl.ch

Martin Odersky
EPFL, Switzerland
martin.odersky@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. Its modular design allows users to extend Jet with modules that produce good performing code. Through a modular code generation scheme, Jet can generate programs for both Spark and Hadoop. Compared with naïve implementations we achieve 143% speedups on Spark and 126% on Hadoop.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages

General Terms

Languages, Performance

Keywords

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

BigData '12, August 31. 2012, Istanbul, Turkey

1. INTRODUCTION

Over the course of the past decade, numerous systems for cluster computing on big data have been presented [10, 25, 20, 24, 12]. These systems' programming model impose a trade-off between generality, performance and productivity. Systems like Hadoop MapReduce [3], and Dryad [14] provide a low level general purpose programming model that allows one to write fine grained and optimized code. However, these low level optimizations greatly sacrifice productivity [8]. Models like Spark [12], FlumeJava [8] and Dryad/LINQ [25] 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 a lack of knowledge about the program structure.

Domain-specific approaches, like Pig [20] and Hive [24], have a narrow, side-effect free and domain-specific interface that allows one to be both productive and to benefit from relational optimizations with them. However, they come with their own set of limitations. Their programming model is often too specific for a wide range of problems. This requires one to write their own functions for general problems, which is cumbersome to do and again hard to optimize, and is otherwise a departure from the original programming model. Moreover, there is the high overhead of learning a new language. It is also hard to extend these frameworks with optimizations for a new domain.

Recently there have been several efforts aimed at making programming distributed batch processing efficient, productive and general at the same time. Steno [19] 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 [16] and HadoopToSQL [15] apply byte code analysis, to extract information about unused data columns and selection conditions. The reason for doing this is 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 is built upon language virtualization [18] and lightweight modular staging [21] (LMS), and has the same syntax and semantics as the standard Scala programming language, with only a few restrictions. Because Jet includes common compiler optimizations and relational optimizations, as well as domain-specific optimizations, it produces efficient code which closely resembles hand optimized implementations. Furthermore, 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, which 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. It takes the whole program into account, and can safely apply optimizations without being overly conservative due to the available effect information.
- We show that Jet allows easy language extension and code portability for distributed batch processing frameworks.

In Section 2 we provide background on language virtualization, LMS and distributed batch processing frameworks. Then, in Section 3 we explain the programming model and present simple program examples. In Section 4 we explain the novel projection insertion optimization algorithm in Section 4.1 and Section 4.2 explains the fusion optimization. We evaluate Jet in 5 and discuss our approach in Section 6. Jet is compared to state of the art in Section 7, future work is in Section 8 and we conclude in Section 9.

2. BACKGROUND

In this section we explain language virtualization in Scala, Lightweight Modular Staging (LMS) library [21, 22], 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 [18] 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 is 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: `jetDSL{ \\ dsl code }` gets translated into: `new JetDSL { def main(){...}}` This makes all the DSL functionality defined in JetDSL visible in the body of the by name parameter passed to the jetDSL method. Although modified, Virtualized Scala is fully binary compatible with Scala and can use all existing libraries.

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 the 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 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 effect 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, so it can be invoked. If the compilation

[1] Scala's support for multiple inheritance

```
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))
  }
}

// code generator for println IR
trait PrintlnGen extends ScalaGen {
  def emit(node: Rep[Any]) = node match {
    case PrintlnNode(str) =>
      println("println("+str+")")
  }
}
```

Listing 1: Example of the DSL module used for printing strings.

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 1, we show a simplified version of a DSL module for printing strings. In trait `PrintlnExp` we define how the `println` operation builds the IR node. The `reflectEffect` method defines that the `println` 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.

Until now, LMS has been successfully used by Brown et al. for heterogeneous parallel computing in project Delite [6, 23] and by Kossakowski et al. for a JavaScript DSL [13].

2.3 LMS Optimizations

When writing DSLs, the DSL authors 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 statements out of hot loops *ii)* move statements 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 column-oriented storage in databases and it gives great performance and memory footprint improvements.

LMS also provides a very general mechanism for loop 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 1, we present these optimizations on a single example which parses an array of complex numbers and prints only the real parts of them. Step 1a) shows the original program, step 1b) shows how CSE extracts `size` and inlining replaces `parse` and `split` invocations with their bodies. In step 1c) the array x of complex numbers is split into two arrays of floating points. In step 1d) 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 are then removed by DCE.

2.4 Distributed Batch Processing Frameworks

Jet generates Scala code for Crunch [2] and Spark [12]. Crunch uses Hadoop as the execution engine and provides a collection-like interface, backed by a MSCR implementation as presented by Dean et. al. [8]. Crunch is implemented in Java and provides a rather low level interface, the user must for example specify the serialization for user classes.

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 compared to Hadoop. Like Crunch it provides a collection-like interface, but it is more high-level. Spark has support for multiple serialization frameworks and it also features a REPL 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 1. 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 methods for transforming the data by applying the argument function. These 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 pair. `reduce` is used for general aggregation after the `groupByKey`, `join, cogroup` and `cross` are different types of relational joins. `sort` sorts the data set, `partitionBy` defines partitioning among machines, and `cache` signals that the data should be kept in cluster memory for faster future accesses. Two `DColls` 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.

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

Listing 2: An example of a simple word count program.

In Listing 2, we show an implementation of a simple word count example. We notice that the code does not show any `Rep` types. Since a large subset of the Scala library is implemented as a DSL module, functions like `split` and string 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 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 [7] and Twister [11] 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

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]`
	`sortByKey(asc: Rep[Boolean])`	`DColl[(K <: Ordered, V)] => DColl[(K, V)]`
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.

liveness of all fields it can possibly remove. We define such a liveness analysis for each operation in our programming model, we identify rules on how one operation influences the liveness of fields. For operations that have a closure parameter, all optimizations for structs present in LMS are applied, and we can analyze the code after DCE has eliminated unused fields.

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. In 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 2 shows the tree of nested fields for the class `Tuple2[String, A]`. The nodes describe the class of a parents 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]
```

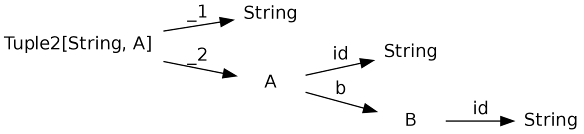

Figure 2: Visualization of the tree of fields for class Tuple2[String, A].

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. This primitive can for example be used to create the set of access paths needed for a `save` operation. For the class `A`, it returns the access paths $id, b, b.id$.
- *Closure analysis*: This primitive analyzes a closure and returns a set of all access paths relative to that 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. The optimizations in LMS ensure 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.

Operation	Access path computation and propagation	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.

Table 2 shows how relevant operations in Jet are analyzed and how access paths are propagated. The first column is the operation name, and the last column shows whether we treat this operation as a barrier. In the middle column we describe the rules for propagation of access paths. 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 access paths.

4.2 Operation Fusion

In Section 3 we have shown that the `DColl` class 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 2.3. 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 [19] for each data record processed. To reduce this overhead and to provide more opportunities for compiler optimizations we have implemented fusion of the monadic operations `map`, `flatMap` and `filter` through the underlying loop fusion algorithm described in Section 2.3.

The loop fusion optimization described in Section 2.3 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. Alternatively, we could have taken the path of Murray et al. in project Steno [19] by generating an intermediate language which can be used for simple fusion and code generation. Also, we could use the Coutts et al. [9] approach of converting `DColl` to streams and applying equational transformation to remove intermediate results. After implementing the algorithm by reusing the LMS loop fusion we are confident that it required significantly less effort than other alternatives.

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 flow 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 representations. We show the program translation for this transformation in Listing 3. 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 fusion 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 `DColls`. This would be possible for Hadoop, as it supports multiple outputs in the map phase but is not currently supported in Spark 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 evaluation, we concluded that the gain is not significant compared to using back-end with a higher-level programming model. Therefore, as an alternative, we used the Crunch framework.

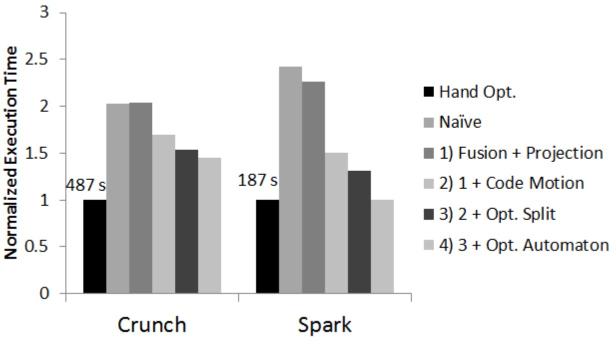

Figure 4: The Word count benchmark.

Unlike MapReduce based back-ends, Spark's design uses the pull data-flow model, implemented through iterators. When generating code for the pull data-flow model from the loop based model (push data-flow) we had to insert a buffer from which data can be pulled by an iterator.

The overall effect of the fusion optimization is that it removes all superfluous method calls and data structures from the program. Additionally, fused operations provide more opportunities for other compiler optimizations. This results in generated programs which are very close to the hand optimized versions, although they were written in a high-level declarative way. To our knowledge there are no distributed batch processing frameworks that achieve this today.

5. EVALUATION

We evaluate the optimizations by comparing the performance of three programs executed with different optimizations enabled: *i)* a word count with prior text processing, *ii)* TPCH [5] query 12 and *iii)* a k-means application. To evaluate the extensibility of the framework we introduce a new Jet module 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 they have a 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. We used Whirr 0.7.1 [1] to set up the cluster, ensured that both hard drives were used for the distributed file system and set the setting to reuse the JVM for multiple map tasks. For benchmarking Spark we used Spark version 0.5.0 for the tests, and started the cluster using Spark's EC2 script. For Spark we had to tweak settings to ensure that the programs run correctly, including increasing the amount of available memory to 6 GB and setting the parallelism to a value found after experimentation.

While doing preliminary benchmarking we found some simple optimizations 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, based on the regular expression pattern we select between Java's implementation and the *dk.brics.automaton* library [17].

For serialization of data we used Jet to generate specialized code to achieve minimal overhead for Crunch. 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's compiler optimizations without focusing on projection insertion. We chose 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 a naïve 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 also use the optimized automaton regular expression library.

Our input is a 124 GB set of plain text version of Freebase Wikipedia articles. The program uses five regular expressions to remove words of the input that were not parts of the plain text in the article. This benchmark does not benefit from projection insertion but we include it for the comparison with the Pig framework in Section 5.3.

In Figure 4 we show the job times for these versions normalized to the hand-optimized program version. Compared to the naïve version, the performance improvements of all optimizations combined range from 40% for Crunch to 143% for Spark. The base performance of the frameworks differ by a large margin for this benchmark. 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.

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 evaluate all optimizations separately and all of them together and compare it to the naïve version. As the data set we use 200 GB plain text input generated by the DbGen tool [5].

In Figure 5 we show job times for different optimizations normalized to the naïve program version on different frameworks. We notice that projection insertion gives 30% percent better performance on Crunch while on Spark the projection insertion improves the performance by 35%. Overall our optimizations show speedups of 126% on Crunch and

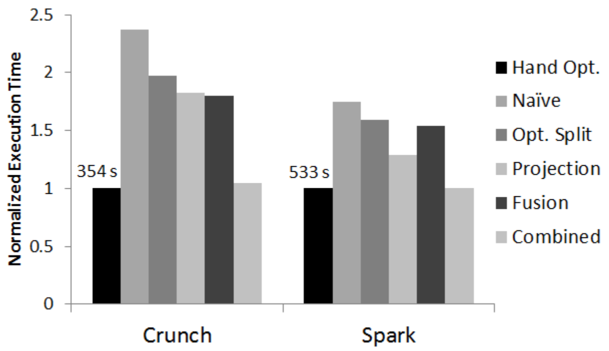

Figure 5: TPCH query 12 benchmark.

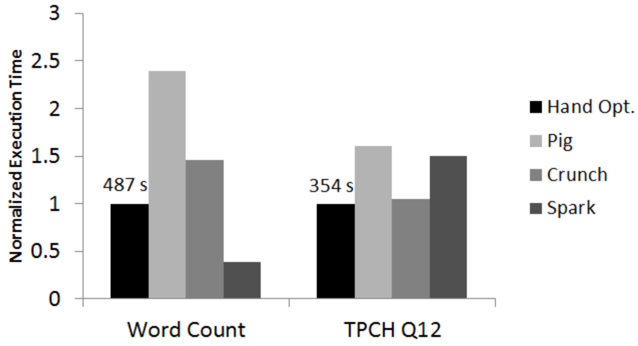

Figure 6: Comparison between the hand-optimized Hadoop versions, Pig, Crunch and Spark.

75% on Spark over the naïve version, and in this benchmark our fastest version is almost as fast as the hand-optimized Hadoop version (5% slower).

5.3 Comparison with Pig

In Figure 6 we compare the hand-optimized Hadoop and Pig programs with our most optimized program versions targeting Crunch and Spark. The y-axis is normalized to the hand-optimized Hadoop and the overall job time is stated above the bar.

We notice that our generated Crunch programs are faster than Pig by 53% on TPCH query 12 and 65% on Wordcount. For the word count benchmark even the naïve version is faster than Pig by 18%, we believe this to be due to the high overhead of Pig's String operators. We further see that for TPCH query 12, the performance of our program version is almost as fast as the hand-optimized Hadoop version.

For the sake of showing a comparison between the Hadoop based frameworks and the Spark framework we include the Spark results in the graph. Spark performs better for the word count example, and performs worse than Crunch in the TPCH benchmark. We are uncertain about the exact causes of this behaviour. However, code portability of Jet can be used to select the appropriate execution engine for each program yielding even better performance improvements for a specific program.

5.4 Extensibility

To evaluate the extensibility of Jet we decided to extend it in two ways: with a high-performance abstraction for the k-means benchmark (Vector) and with a new code generator that targets the Crunch framework.

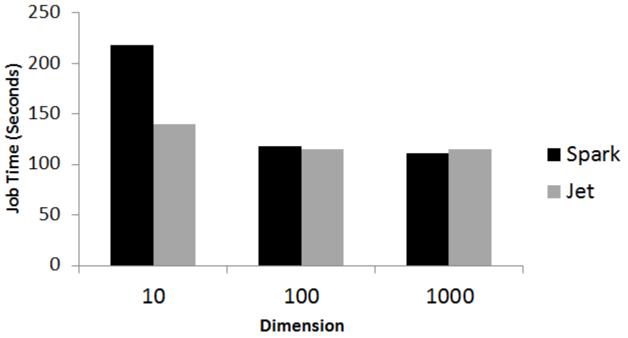

Figure 7: k-means benchmark.

The Vector module for k-means. We took a version of Spark's k-means [12] application and ported it to Jet. This application can neither benefit from projection insertion nor from operation fusion. The Spark code makes use of an abstraction for multi-dimensional points called `Vector`, which uses iterators. We added a `Vector` module to Jet which uses compiler support to generate efficient loops instead of iterators. The implementation of the `Vector` module is 100 lines long and took one man day.

We evaluate the performance of our version against the original Spark version. Because Spark is faster than Hadoop for iterative programs, we only evaluate this benchmark on Spark. As input we used synthetic data with 10 to 1000 dimensions, 100 centers and we keep the $dimensions * points$ factor constant so that each input file is around 20 GB.

Our results are similar to those described by Murray et al. in [19]. In lower dimensions, our optimization shows significant speedups, while for 1000 dimensions, performance is slightly worse. We believe that the iterator overhead is considerable when there are 10 dimensions, because after iterators are removed, performance is much better. At higher dimensions it is possible that the JVM can do a better job of optimizing if the code is smaller, since we observe that our pre-optimized and larger code becomes slightly slower. Regardless, our implementation still performs more consistently for different dimensions.

The Crunch module. Initially we planned to support Hadoop by generating code for the Scoobi [4] framework. We were unsatisfied by its performance however, which lead us to implement code generation for the Crunch framework. The code for this module is only 300 lines long and the implementation took four 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:

- Subtype polymorphism (for user defined classes) is currently not supported.

- Jet can only apply optimizations on functions provided as a DSL module for LMS.

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 of the task. For the benchmarks we tested this process takes between 4 and 14 seconds. Although this can be significant, it needs to be done only 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.

In cases where the data processing time is comparable to the user reaction time, the optimizations we perform are not needed. We could implement a version of Jet that does not perform staging but instead executes the original code straight away. This feature is not implemented in Jet, but Kossakowski et al. [13] have done this for a 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 such parameters, but in the future work we want to introduce a configuration part of the DSL to unify the configuration of different back-ends.

7. RELATED WORK

Pig [20] 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, Pig can not profit from the rich ecosystem of productivity tools for Java.

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 extensible.

Steno [19] 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 projection insertion. The drawback of Jet is that it is slightly limited in generality.

Manimal [16] and HadoopToSQL [15] 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. However, 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 projection insertion. In future work we plan to introduce early filtering which will push filter operations in front of expensive operators that require a barrier. Also, we plan to include a program analysis phase which will allow building of indexes.

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

Jet currently only operates on distributed data sets so programs written in it can not be used for in-memory data. We plan to integrate Jet with the Delite [6] 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 multi-core systems and GPUs.

9. CONCLUSION

We have presented the distributed batch data processing framework 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 and Crunch. Presented optimizations result in speedups of up to 143% in Spark and up to 126% in Crunch.

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

10. REFERENCES

[1] Apache Whirr, http://whirr.apache.org/.

[2] The Crunch framework, https://github.com/cloudera/crunch.

[3] The Hadoop framework, http://hadoop.apache.org/.

[4] The Scoobi framework, https://github.com/nicta/scoobi.

[5] TPC Benchmark^{TM} of the Transaction Performance Council, http://www.tpc.org/tpch/.

[6] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, page 89–100, 2011.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. *ACM SIGPLAN Notices*, 45(6):363–375, 2010.

[9] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, volume 42, page 315–326, 2007.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 810–818, New York, NY, USA, 2010. ACM.

[12] M. Z. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI*, 2012.

[13] T. R. Grzegorz Kossakowski, Nada Amin and M. Odersky. Javascript as an embedded DSL. In *ECOOP*, pages 409–434, 2012.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, page 59–72, 2007.

[15] M. Iu and W. Zwaenepoel. HadoopToSQL: a mapReduce query optimizer. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, page 251–264, New York, NY, USA, 2010. ACM.

[16] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.*, 4(6):385–396, Mar. 2011.

[17] A. Møller. dk. brics. automaton–finite-state automata and regular expressions for java, 2005.

[18] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, page 117–120, 2012.

[19] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, page 121–131, New York, NY, USA, 2011. ACM.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, page 1099–1110, 2008.

[21] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, page 127–136, 2010.

[22] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012.

[23] T. Rompf, A. Sujeeth, H. Lee, K. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. *Arxiv preprint arXiv:1109.0778*, 2011.

[24] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996 –1005, Mar. 2010.

[25] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, page 1–14, 2008.