ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Jet: An Embedded DSL for Distributed Data Parallel Computing

Master Thesis

Stefan Ackermann

July 16, 2012

Advisors: Prof. Dr. M. Odersky, Prof. Dr. P. Müller, MSc. V. Jovanovic

Department of Computer Science, ETH Zürich

Abstract

Distributed Data Parallel computing has been simplified drastically with the introduction of MapReduce [13], but MapReduce programs are rather tedious to write. Many frameworks have been created to make MapReduce programming simpler, yet efficient. These approaches either are libraries that can not apply optimizations due to missing information about the code, or external DSLs, which require a significant effort to define and implement and sacrifice generality.

This thesis presents Jet, a DSL embedded in Scala that uses the complete program information to apply complex optimizations like Projection Insertion, Code Motion and Loop Fusion. Jet only has a few restrictions compared to regular Scala, and it can apply optimizations across all supported language constructs such as conditionals, loops and functions. The programs we tested performed between 33% and 174% faster with our optimizations.

Contents

Chapter 1

Introduction

Websites today deal with an increasing amount of user data and analyzing this data has become an important problem for companies. The analysis enables recommendation systems or is even needed for business logic like billing or security surveillance. As storage space becomes cheaper, the amount of stored data has increased. The analysis of these huge data sets can not be done on a single computer, and programming on distributed systems is arguably more complex, especially when dealing with failures.

Databases are used to separate data management and business logic. However, guarantees provided by databases like atomic commits do not scale well, and are therefore not considered for huge data sets. Additionally, databases work with strictly relational data, which makes them unsuitable for plain text log analysis for example.

Functional abstractions have been proposed to separate the issues of fault tolerance and scalability from the actual logic of the program [13]. This thesis focuses on distributed data parallel computing frameworks, which provide simple, scalable, and fault tolerant batch processing by restricting the programming model to data parallel functions.

Many of these systems have been studied in the past decade. These systems are all restricted by the general trade-off between generality, performance and productivity. Dryad [15] and MapReduce [13] have low level programming models in general purpose programming languages, so that they allow fine grained control over the execution of the program. This however limits the productivity. Systems like Spark [29], FlumeJava [9] or Dryad/LINQ [28] allow a high level programming model in a general purpose language, but they rely on high level abstractions that limit the performance. To apply relational optimizations, the whole program must be analyzed, therefore none of the aforementioned frameworks can apply those automatically.

DSL's like Pig [24] or Hive [27] define their own restricted language, for which they do have all the program information available. Generality is provided by the possibility of adding user defined functions in a general purpose programming language, but doing so will break optimizations, as these systems will not analyze these.

Another approach is to use general and productive tools, and then apply optimizers that work on the compiled program. Manimal [17] and Hadoop-ToSQL [16] can work with any Hadoop [6] framework and do have the complete program information available to apply relational optimizations. However, these tools need to recreate the intentions of the programmer from the most low level representation of a program. This leads to missed optimization opportunities due to information lost or obfuscated during compilation.

This thesis presents a new domain specific language for distributed data parallel programming called Jet. Jet has the same syntax and semantics as regular Scala with a few restrictions. It provides a high level, declarative interface similar to Spark, and it applies compiler optimizations and domain specific ones to generate highly performant code. Jet is designed in a modular and extensible way, allowing to add new modules for specific programs. The code is also portable, the same program can be compiled for use on Hadoop or on Spark.

Jet makes the following contributions to the state of the art:

- We implement and open-source the Jet framework for distributed data parallel computing. Jet features a high level programming model and applies domain specific and compiler optimizations, even over user defined functions.
- We present a novel, elegant Projection Insertion algorithm that is conceptually easy, but works with general program constructs such as classes, conditionals, loops and methods by reusing the facilities provided by the underlying framework. It does not miss opportunities due to the high level information available about the program, which includes effects.
- We show that the optimizations and the language for Jet are extensible and modular, and how we use that modularity to provide code portability.

Chapter 2

Background

This chapter will present the projects Jet builds upon. We will explain the relevant components of LMS [25], the basis for this project. We will also introduce MapReduce and the frameworks that we generate code for.

2.1 LMS

LMS, Lightweight Modular Staging, is a Scala framework for writing Domain Specific Languages (DSL). It uses a special version of the Scala compiler and library named Scala Virtualized [19]. Following we will introduce first Scala and Scala Virtualized and then explain LMS.

2.1.1 Scala

Scala [22] is a general purpose programming language, that allows concise and high level programming. It features a powerful type system, and integrates features from both object orientated and functional programming. It executes on the JVM, and allows to reuse Java libraries.

Scala has support for limited multiple implementation inheritance through Scala's traits [23]. Scala's traits are similar to Java interfaces, but they also allow functions to be implemented. Multiple traits can be mixed into the same class. When traits are mixed together, the order of the composition defines the method lookup order of the dynamic dispatch. Mixin composition is used in LMS and Jet extensively, all modules are implemented as traits, so that they can be combined freely.

Scala's powerful type system also has support for implicit conversions. If an object of type `A` is found at a location where a type `B` is needed for type checking to succeed, the Scala compiler will search for an implicit method that can convert an object of type `A` to an object of type `B`. A call to that method is then inserted, and the type checker accepts the code. This allows to hide implementation details, it is for example used to choose appropriate converters for classes with generic types.

2.1.2 Scala Virtualized

Scala Virtualized [19] is an extension to the Scala language and compiler with a small number of features to allow deep language embedding. Deep embedding enables a DSL to reuse the Scala parser and type system and other tools for the regular Scala language. Scala Virtualized provides embedding by calling overloadable methods for all regular language constructs like conditionals, loops, variable declarations and pattern matching. The DSL can then use this call to create an intermediate representation of that language construct. For example, for the code `if (c) t else e`, Scala Virtualized will call the method `__ifThenElse(c, a, b)`. The DSL can then create an intermediate representation (IR) node for this conditional. This process is called staging.

2.1.3 LMS

Jet is based on Lightweight Modular Staging (LMS), a framework for writing DSLs. LMS uses the features of Scala Virtualized to provide a modular compiler infrastructure for developing staged DSLs. It uses a special marker type, `Rep[T]`, to distinguish between staged values and unstaged ones. A value with type `Rep[T]` is used for staging, optimizing and generating code to represent type `T` in the next stage. The interface for the DSL is defined by specifying the methods available for a certain concrete type of `Rep[T]`, and these methods are use for building the intermediate representation (IR).

All the staged values in the DSL code need to have their type wrapped in `Rep[T]`. This seems rather intrusive at first, but Scala's type system often hides this fact. During the development of code using Jet, the `Rep` types were only visible in type annotations in the parameter list of functions and for Scala's tuples.

LMS is designed in a very modular way, which allows the DSL developer to compose the interface, optimizations and code generation of the DSL freely. LMS contains interface and code generation for most language constructs in the regular Scala language, for example methods on primitive types, essential data structures, and conditionals and loops. These can all be enabled separately by mixing in the corresponding traits. These components all have corresponding code generators implemented to generate C, CUDA or Scala code. In Jet, we only use the Scala code generators.

LMS additionally includes effect tracking. All effects of operations have to be declared by the DSL developer, the provided modules in LMS do contain these. LMS uses the effect information to apply optimizations in many cases whereas a general purpose compiler has to make the worst case assumptions about side effects of statements. LMS can safely apply Code Motion and dead code elimination on pure code, it only has to ensure that effectful statements do not get reordered in respect to other effectful statements.

Listing 1 shows all the code involved in creating a simple DSL module with the convention used in LMS. The module can be used like a simple logger: When the method `report_time` is called, code is generated that writes the argument together with the current time stamp. The interface has to be defined (Line 2), the internal representation (IR) node created (Line 5), the IR node defined (Line 7), the mirroring – used for updating the IR when transformations are applied – (Line 9) and the code generation for Scala (Lines 11–13). Note that this is an effectful operation, because it has an observable side effect, the printing. For this reason we call `reflectEffect` which makes the effect information explicit for LMS.

2.1.4 LMS Optimizations

LMS comes with a wide array of classical compiler optimizations, such as dead code elimination (DCE), Code Motion, common subexpression elimination, function inlining and loop fusion. It can use these optimizations aggressively due to the available effect information. Additionally, the DSL author can implement domain specific optimizations.

LMS also contains optimizations for struct-like objects, objects which contain a list of fields and user defined methods. These objects can be used like normal objects, but in the generated code they will only appear if they are required, otherwise LMS works with the fields of the objects directly. If the object is not required, some fields of the objects may never be read or printed. In that case, these fields do not have to be computed.

Loop Fusion is handled very generically for LMS. LMS supports both vertical fusion (consumer and producer) and horizontal fusion (loops with the same range). Besides reducing the number of loops, the merging of scopes gives LMS more opportunities to apply its other optimizations.

Listing 2.1 shows how these optimizations are applied to a simple piece of code. The code in Listing 2.1a parses strings with persons and then prints the names of all persons who are older than 100 years. LMS will inline all the methods, and will fuse loops, resulting in the code in Listing 2.1b. Then the

```
// interface definition
def report_time(x: Rep[String]): Rep[Unit]

// override method to create IR
override def report_time(x: Exp[String]) =
  reflectEffect(ReportTime(x: Exp[String]))

// IR node
case class ReportTime(x: Exp[String])
        extends Def[Unit]

// mirroring
def mirrorDef( ... { // simplified
  case ReportTime(x) => ReportTime(f(x))
}

// code generation
def emitNode( ... { // simplified
  case ReportTime(x) => stream.println(
    """println(System.currentTimeMillis+" "+%s)"""
    .format(quote(x)))
}
```

Listing 1: Simple DSL module following LMS best practices for printing a line including the current time stamp.

aforementioned struct optimizations are applied, the `Person` class is treated like a struct. In this case, the optimization determines that the fields are read from a constructor invocation, and can then bypass the constructor to read these values directly. This results in the code in Listing 2.1c. Then the invocation of the constructor becomes dead, because there is no statement reading from it left. Because LMS knows that the constructor invocation does not have any side effects, it can safely remove it. The same goes for the parsing of the `email` and `zipcode` fields. Code Motion can then move the parsing of the name into the conditional. After these optimizations, the final optimized code is shown in Listing 2.1d.

2.2 Distributed Data Parallel Programming

As websites become much more dynamic and handle more user content, Distributed Data Parallel Programming – also called Big Data – has become more popular than ever. MapReduce [13] has shown a way of dealing with

```
def parsePerson(s: Rep[String]) = {
  val f = s.split("\t", 4)
  val name = f(0)
  val age = f(1).toInt
  val zipcode = f(2).toInt
  val email = f(3)
  new Person(name, age, zipcode, email)
}
val persons = for (x <- lines)
  yield parsePerson(x)
for (x <- persons)
  if (x.age > 100)
     println(x.name)
```

(a) Original program

```
for (x <- lines) {
  val f = s.split("\t", 4)
  val name = f(0)
  val age = f(1).toInt
  val zipcode = f(2).toInt
  val email = f(3)
  val person = new Person(name, age, zipcode, email)
  if (person.age > 100)
    println(person.name)
}
```

(b) Inlining and Loop Fusion

```
for (x <- lines) {
  val f = s.split("\t", 4)
  val name = f(0)
  val age = f(1).toInt
  val zipcode = f(2).toInt
  val email = f(3)
  val person = new Person(name, age, zipcode, email)
  if (age > 100)
    println(name)
}
```

(c) Struct optimizations

```
for (x <- lines) {
  val f = s.split("\t", 4)
  val age = f(1).toInt
  if (age > 100) {
    val name = f(0)
    println(name)
  }
}
```

(d) Code motion and dead code elimination

Figure 2.1: Step by step optimizations in LMS

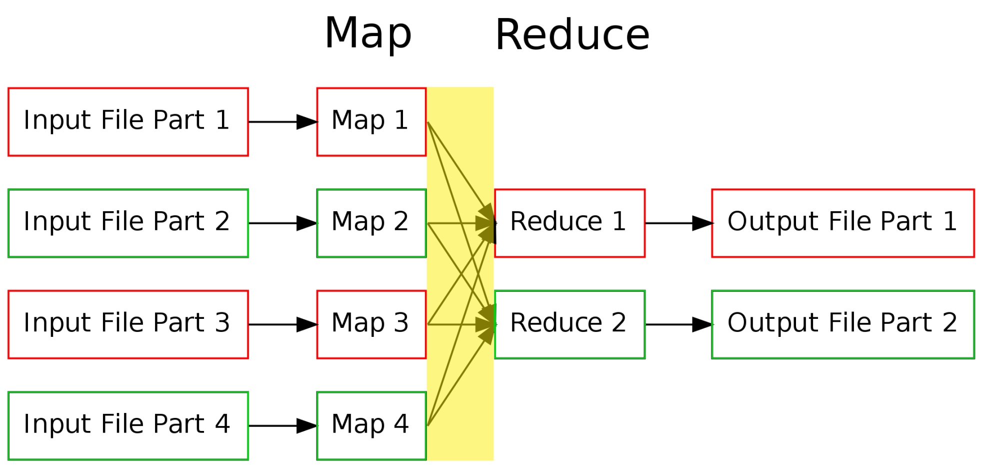


Figure 2.2: MapReduce execution example. Red and green denote machine assignments, the shuffle phase is highlighted yellow.

Big Data, and this programming model has since then persisted. We will introduce the MapReduce ideas and the derived projects used in Jet.

2.2.1 MapReduce

MapReduce is a paradigm for programming computations involving many computers. It has been shown to scale to thousands of computers easily. In MapReduce, all computations are expressed as a series of *map* and *reduce* transformations. A *map* function transforms one input key-value pair and returns any number of key-value pairs. These key value pairs are then redistributed over the network, to place all values for the same key on the same machine. The *reduce* function then accepts one key and all its associated values and produces a new key-value pair. In the MapReduce model, the user has to specify the computation as a series of *map* and *reduce* steps. The advantage of the MapReduce is that due to its parallel nature, scalability and fault tolerance is handled by the framework.

Figure 2.2 shows a diagram of a MapReduce execution. The mapper code is executed for each split of the input files on the distributed file system. The data is then repartitioned in the shuffling phase, the key in the key-value pair determines to which reducer it is copied. The reducers then run the *reduce* function and produce output files.

Although the MapReduce model simplifies distributed programming by providing fault tolerance and scalability, expressing programs with just *map* and *reduce* is still tedious. FlumeJava [9] describes how a collection like interface can be mapped onto MapReduce, which makes programming simpler. Additionally the authors describe how programs expressed in the FlumeJava programming model can be executed on a MapReduce runtime with a minimal number of expensive shuffling stages.

2.2.2 Hadoop

Hadoop [6] is an open source implementation for MapReduce. It is implemented in Java and contains a distributed file system and a MapReduce implementation.

There are many frameworks using Hadoop as the execution engine. Many frameworks provide a FlumeJava implementation, allowing to use the MapReduce paradigm with higher level constructs such as joins. Others provide workflows, where an update to one file can automatically trigger one job and all its dependent jobs. We generate code for Crunch [11] and Scoobi [21], which both are FlumeJava implementations for Hadoop. Crunch is written in Java; Scoobi is written in Scala and makes use of Scala's powerful type system including implicits, to provide a high level interface which provides easy serialization for user classes and hides other details of Hadoop.

2.2.3 Spark

Spark [29] introduces a new concept called Resilient Distributed Datasets (RDD). The execution of a program is similar to MapReduce, but it keeps objects by default in memory and provides fault tolerance through lineage. Spark was designed to allow easy reuse of data for iterative jobs, and the authors showed that it can be up to 30x faster than Hadoop for these jobs. It is however not limited to iterative jobs, it can handle any program that Hadoop can. Spark is implemented in Scala.

Chapter 3

Solution: Jet, An Embedded DSL for Distributed Data Computing

We present Jet, a DSL for distributed data parallel computing. Jet has a high level, declarative interface, which allows it to apply compiler and relational optimizations. We focus on compile-time optimizations, which are impossible or hard for libraries without compiler support to implement. Jet is written in a modular way, is extensible and compiles to multiple big data runtimes.

Jet is based on LMS, as introduced in Section 2.1.3. It compiles from Scala code using our DSL to Scala code that interfaces with one big data runtime, the target. Our targets are, as of this writing: Crunch and Scoobi, FlumeJava [9] implementations that run on Hadoop, and Spark.

We set the following goals for Jet:

- Conciseness: High level and type safe code
- Generality: Support loops and conditionals, similar to regular Scala code
- Fast: Use compiler optimizations to generate highly efficient code
- Modularity: Enable reusability, allow more operations for targets
- Portable Code: Code stays portable between different big data runtimes
- Extensible: Enable program specific extensions

Operation	Transformation
`DList(uri: R[String])`	`String => DList[T]`
`save(uri: R[String])`	`DList[T] => Unit`
`map(f: R[T] => R[U])`	`DList[T] => DList[U]`
`filter(f: R[T] => R[Boolean])`	`DList[T] => DList[T]`
`flatMap(f: R[T] => R[It[U]])`	`DList[T] => DList[U]`
`groupByKey()`	`DList[(K, V)] => DList[(K, It[V])]`
`groupByKey(p: R[Partitioner])`	`DList[(K, V)] => DList[(K, It[V])]`
`reduce (f: (R[V], R[V]) => R[V])`	`DList[(K, It[V])] => DList[(K, V)]`
`cogroup(right: R[DList[(K, W)]])`	`DList[(K, V)] => DList[(K, (It[V], It[W]))]`
`join(right: R[DList[(K, W)]])`	`DList[(K, V)] => DList[(K, (V, W))]`
`++(other: R[DList[T]])`	`DList[T] => DList[T]`
`partitionBy(p: R[Partitioner[T]])`	`DList[T] => DList[T]`
`takeSample(p: R[Double])`	`DList[T] => It[T]`
`materialize()`	`DList[T] => It[T]`
`cache()`^{1}	`DList[T] => DList[T]`
`sortByKey(asc: R[Boolean])`^{1}	`DList[(K, V)] => DList[(K, V)]`
`sort(asc: R[Boolean])`^{2}	`DList[T] => DList[T]`

Table 3.1: DList operations. For space reasons, `It` represents the Scala `Iterable`, `R` the `Rep`. The `Rep[_]` types in the right column are omitted. ^{1}: Only supported on Spark. ^{2}: Only supported on Crunch.

3.1 Programming Model

Jet offers a collection class `DList[T]`, which represents a collection with element type `S <: T` distributed over multiple machines. DLists can only be transformed or persisted as a whole, individual elements can not be updated. Jet has a high-level interface by providing higher-order functions and relational operations, similar to the functions on Spark's [29] `RDD` type. All the provided functions are data parallel.

We define operations for creating and persisting, for collecting all elements on the driver program, for transforming all elements and for aggregating the elements to one value for `DList`'s. Some operations are just available for a certain target, we make those available in an extended version of Jet for that target. The operations are summarized in Table 3.1. We use Scala's

```
1  def describe(t: Rep[(Int, Person)]) =
2    t._2.name + " is the oldest person in " + t._1
3
4  def oldestPersonPerCity(x: Rep[Unit]) {
5    val persons = DList("persons").map(Person.parse(_, "\t"))
6    persons
7      .map(p => (p.zipcode, p))
8      .groupByKey()
9      .reduce((p1, p2) => if (p1.age > p2.age) p1 else p2)
10     .map(describe)
11     .save("oldestPersons")
12 }
```

Listing 2: Example program

tuples for key value pairs.

Operation `DList()` creates a new `DList`, and `save` saves one. The monadic operations `map`, `filter`, and `flatMap` transform each element into 0 or more new elements. To group all the elements with the same key together, `groupByKey` is used, with or without partitioner. To reduce all elements with the same key, `reduce` can be used. These operations cover the model of MapReduce, as described by Dean et al. [13].

We support additionally the relational operations `cogroup` and `join`. We support concatenation with the operation ++. To redistribute elements among the machines, `partitionBy` can be used. Sampling is supported by `takeSample`. To collect all elements on the driver, `materialize` can be used.

For Spark, we also allow the use of `sortByKey`, which needs the key to be a subtype of `Ordered`. If a collection in Spark is used multiple times, `cache` will try to keep it in memory. For Crunch, we allow the `sort` method which sorts based on the objects serialization.

We include a sample program in Listing 2. This program first parses the entries in a `DList[String]` as `Person` objects. It then proceeds to print out the name of the oldest person for each city. Note the user defined function on Line 1.

3.2 Implementation

Jet uses a sequence of steps for the compilation, as shown in Figure 3.1. The first step is staging, during which we create our intermediate representation (IR). Then we apply rewrites specific to each framework, e.g. the merging of `groupByKey` and `reduce` for Spark, because this enables the use of local

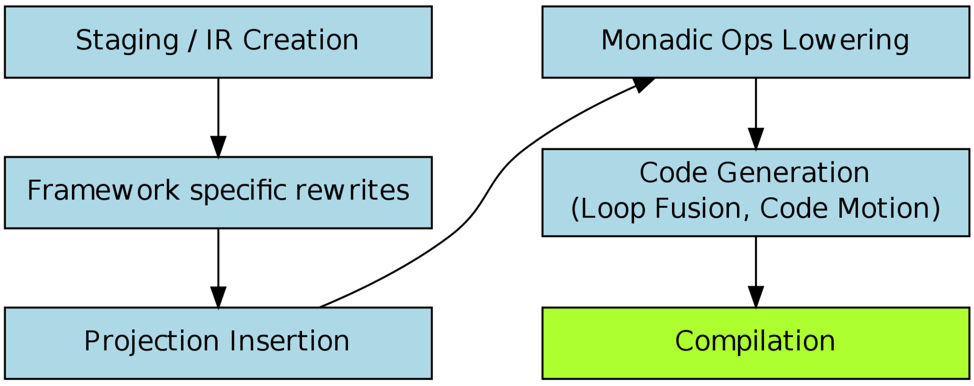


Figure 3.1: Steps inside the compiler

aggregation. Next we run Projection Insertion to remove unneeded fields from objects. We do this by inserting additional `map` operations that only keep the needed fields, or by changing existing `map` operations that output an object. Then we transform the monadic operations to loops, to allow Loop Fusion to fuse them later. Next we generate the code, during which LMS applies Loop Fusion and Code Motion. We end up with Scala code for use in a certain framework, and we start the compilation of that Scala code in the last step. The last step is therefore outside of Jet.

Jet is implemented with the cake pattern, as LMS itself. This means that it is easy to reuse components like analysis and transformations of code and type factories for creating suitable types for a serialization framework. Additionally, it allows each framework to define additional operations.

We allow the user to define his own classes. They have to be specified in a very simple format, as shown in Listing 3. We use a script to generate all the necessary code for its use in Jet. The user can then work with these classes as with regular classes, with full compiler and editor support.

```
class Person(val name: String, val age: Int,
    val zipcode: Int, val email: String)
```

Listing 3: Example class

3.2.1 Modularity and Extensibility

We use the cake pattern, that is also used for the Scala compiler [23]. This allows us to compose our code generators by mixing together different pieces. Per target, we only need a few hundred lines of specific code. To define a target, the following is required:

- Define the code generation for the minimal set of operations
- Define which transformations to apply
- Mix in an appropriate serialization scheme
- If the target supports special operations, then define for each of those:
  - The interface
  - The code generation
  - The access path analysis (see Section 3.3.3)
  - If it is a barrier, how to insert a narrower before

Even though Jet is mainly implemented as a compiler, we can just as easily implement an operation in the target framework code and link against it from the generated code. We use this multiple times, to enable simpler code generation and to program parts with less constraints on performance in a simpler way.

Jet can also be extended with specialized constructs for specific programs. See Section 4.2 for the discussion on the extensibility.

3.3 Optimizations

Our high level description of the program, including effect information, allows us to apply a wide range of optimizations. We will explain what optimizations we can use of LMS in Section 3.3.1. Further we will explain the domain specific optimizations we implemented. We present our regular expression optimizations in Section 3.3.2, the Projection Insertion that removes unneeded fields early in Section 3.3.3, and how we fuse monadic operations in Section 3.3.4.

3.3.1 LMS Optimizations

LMS applies some optimizations by default, as explained in Section 2.1.4. For the example code in Listing 2, LMS only inlines the calls to the function `describe` and removes all statements that do not have an effect (in this example, all data flows are saved, so nothing is removed). For the following discussions, we will only look at the generated code for the lines 7–9 of Listing 2, which prepares the data before the network shuffle.

```
val filtered = in.filter({ s: String =>
  !s.matches("\s+")
})
```

Listing 4: Before regular expression optimization

```
val frontend = new RegexFrontend("\s+")
val filtered = in.filter({ s: String =>
  !frontend.matches(s)
})
```

Listing 5: After regular expression optimization

3.3.2 Regular Expressions

Regular expressions are often used for string processing. In distributed programs, the performance of regular expressions can influence the processing time greatly.

Java supplies methods on Strings directly, which compile a regular expression first and then use it. We rewrite these calls to explicitly create the pattern, and use the pattern for each call. The pattern is often constant, so Code Motion in LMS moves it out of the hot paths.

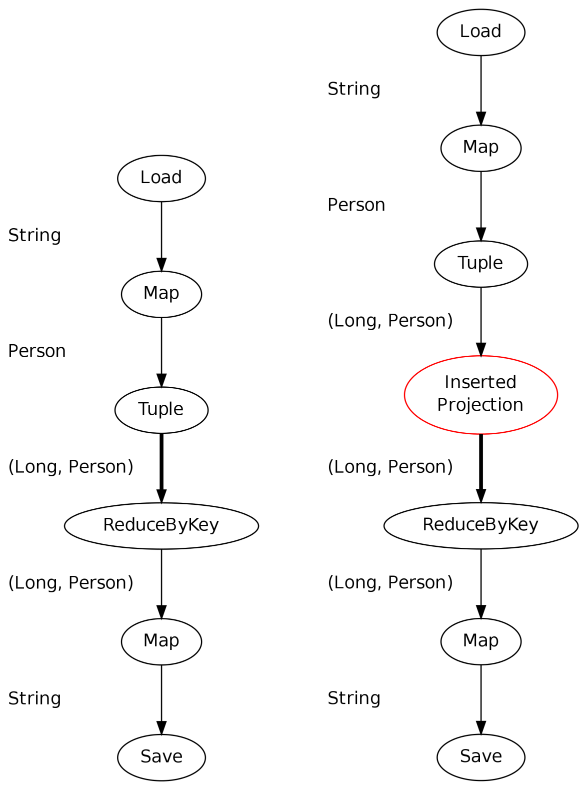
Instead of working with the java builtin regular expression pattern we made our own utility, the `RegexFrontend`. It chooses an appropriate regular expression implementation for each call: It switches to the *dk.brics.automaton* library [18] when possible, and for splits with a single character pattern we implemented an optimized method, as this is a quite common use case. We show how this optimization changes the code in Listing 4 to the optimized version in Listing 5.

These optimizations were not only inspired by Pig [24], which also optimizes splits and makes use of the *dk.brics.automaton* library, we also reused their code. We improved on their scheme even, allowing more regular expression calls to be handled by the faster automaton library.

3.3.3 Projection Insertion

Projection Insertion optimizes a program by removing unneeded values from a computation as early as possible. It is a common optimization in databases and has been described already by Smith. et al. [26]. For distributed programs, this optimization is especially important, as it cuts down the size of the objects which need to be transferred.

We ensure that dead fields are removed before any operation that is sensitive to the amount of fields contained in the objects. Before these operations,



(a) Before narrowing **(b) After narrowing**

Figure 3.2: Dataflow for the example program before and after narrowing. Bold edges are barriers.

which we call barriers, we insert a `map` operation which returns a new object that only contains the live fields. We call this `map` operation the projection or also narrower, because it narrows the type. See Figure 3.2 for the data flow graph of the program described in Listing 2 before and after this optimization.

To insert these narrowers, we need to know which fields are really dead. We can compute the live fields for one operation only if we know the live fields, the fields that are used by some successor of this operation. To ensure that, we analyze one operation at a time in the data flow graph in a reverse topological order.

For analysis of live fields we need a way to describe the live values between operations. Since we support nested types, in Jet, we need to define the fields relative to a type. See Figure 3.3 for the fields nested within `Tuple2[Int, Person]`, which is the type that is used before the barrier. In the figure, the nodes are the types at that path, and the edges are field accesses. The nodes in red denote the fields that need to be alive before the

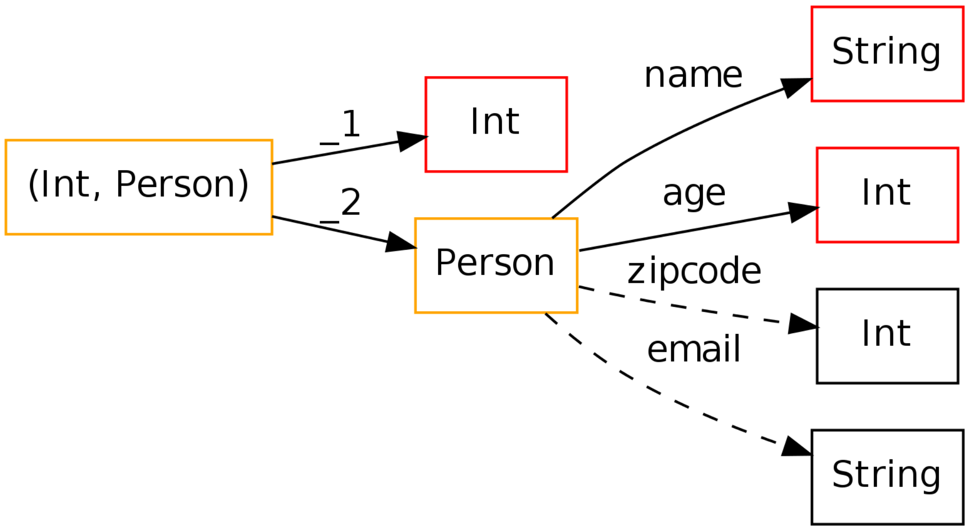


Figure 3.3: Nested fields and access paths for the type in Listing 3

```
val personsNarrowed = persons.map({ in: (Int, Person) =>
  val (key, person) = (in._1, in._2)
  val newPerson =
        new Person_0_1(person.name, person.age)
  (key, newPerson)
})
```

Listing 6: Inserted projection

barrier, and the orange nodes must be alive because one of their children is alive. The narrower, presented in Listing 6, returns an object containing only the red and orange fields.

To describe one field relative to a type, we use the path from the root of the tree to that field. We call this an access path. In the figure, the access paths of interest would be _1, _2.*name* and _2.*age*. To encode the live values between two scopes, we use a set of these access paths.

To compute these access paths for a particular operation, we used the following primitives, combined as explained later:

- *Access paths for a type (ALL)*: This primitive creates access paths for all nested fields within a type. This is needed for all operations that are known to access all fields within a type, like `save`.
- *Closure Analysis (CA)*: This primitive analyzes a closure to collect all the field reads on its input, and returns them as access paths.
- *Rewrite access path (REWRITE)*: This primitive allows to change the prefix of access paths. Since the semantics of several operations are well

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

Table 3.2: Access path computation and propagation for selected operations.

defined and change the type in a way that influences the access paths, we need to update them accordingly. The operation `groupByKey` for example changes the type, and all access paths starting with $\_2.iterable$ need to be rewritten to point to $\_2$.

- *Narrow Closure (N)*: Given a closure and a set of access paths to its output type, this primitive returns a new closure, in which the output only contains the fields corresponding to the given access paths.

To analyze one `map` operation, we first need to narrow the output. If we would not do this, the output symbol would contain dead fields, and to compute these, additional field reads might be performed on the input. Therefore we use the narrow closure primitive to replace the output symbol with one that reads from the old one. If the output symbol of the closure scope is a constructor invocation, LMS will recognize this and read the field values directly. This happens for all fields, and the old output symbol will never be read. It will be deleted by DCE, which in turn will make other parts of the code dead. After we have replaced the output symbol, we can now analyze the access paths the closure needs from its predecessors using the Closure Analysis primitive.

The other operations are treated similarly, as listed in Table 3.2. In the table, P represents the access paths to propagate to the predecessors. S represents the set of access paths as propagated by the successors. $ALL(T, p)$ generates all access paths for the type T with the prefix p, the prefix can be left away to generate all access paths for type T. $R(P, a.x \Rightarrow b.x)$ is the $REWRITE$ primitive, which rewrites all access paths of P matching schema $a.x$ to conform to schema $b.x$. I is the input type of the operation. CA is the Closure Analysis primitive, N is the Narrow primitive.

With this analysis, we can then simply insert projections on any edge in the data-flow graph. Each projection introduces some CPU overhead though, so we only insert them before barriers. In our example, we use the operations which require a network shuffle as barrier, and also the `cache` operation in Spark, which stores the object in memory. In both cases, the performance gained by removing the value is bigger than the performance lost due to the additional work. Additionally to inserting projections, we use the Narrow Closure primitive also on all `map` operations that contain a constructor invocation.

For the projection to work, we also need to generate appropriate types which do not incur overhead for the eliminated fields. We generate specialized code for each field combination that appears in the program. Theoretically there are 2^n possible combinations for a type with n fields, but in reality the number of combinations is bound by the number of edges in the program.

For more information on the specialized code to support our narrowed types, see the elaborations in Appendix A.

3.3.4 Loop Fusion

Our programming model endorses a programming style with many small, higher order functions. Because every method call encurs a small overhead, this may not be the fastest representation of the code. For this reason we[1] transform the monadic operations `map`, `flatMap` and `filter` into a loop representation. The Loop Fusion described in Section 2.1.4 then fuses these loops together, eliminating the overhead of function calls.

With Loop Fusion we also merge the scopes of the functions. This enables LMS to apply all its optimizations – we are especially interested in the optimizations for structs – on a bigger scope. This essentially removes the trade-off described in the previous Section. In the fully optimized code, Code Motion makes the parsing of fields lazy, such that the minimal number of fields is parsed to filter out an element. For further discussion on this, see the explanations for the Figure 4.1.

The targets we support do not offer primitives that can write to multiple outputs from a single mapper. This would be easy to add in Crunch, as Hadoop supports this, but much harder on Spark. We guide Loop Fusion to prevent fusing two loops if this would lead to a loop with multiple outputs.

Loop Fusion is explained in more detail in our workshop paper published on Jet [3].

[1]Loop Fusion was integrated by Vojin Jovanovic

3.4 Summary

We have presented Jet, a DSL for big data computing that supports different targets, produces highly optimized code, and is modular and extensible.

Chapter 4

Discussion

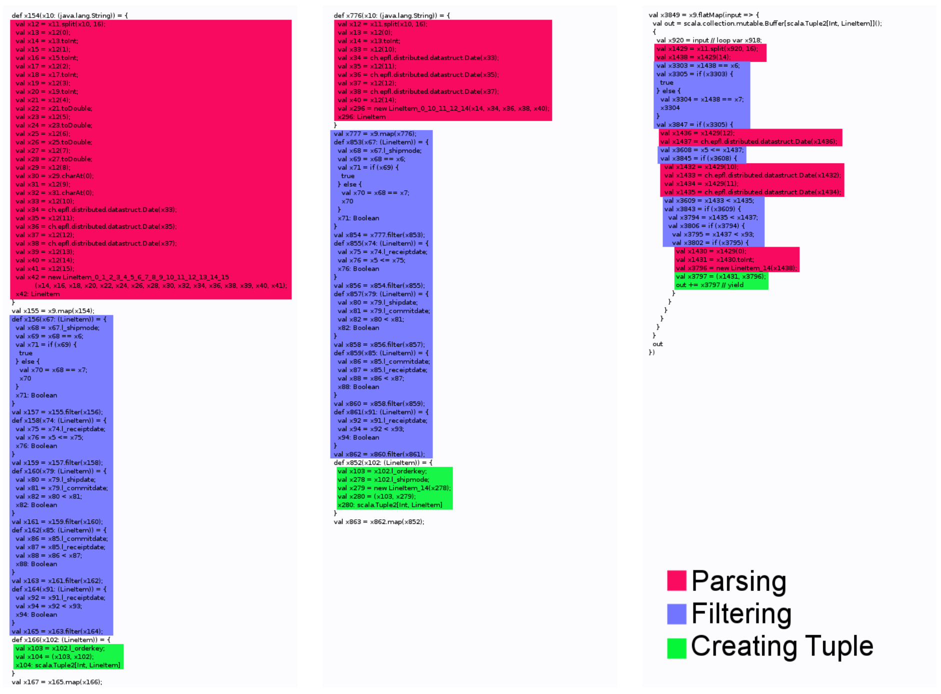
This chapter will discuss the optimizations in Section 4.1, the extensibility in Section 4.2, the effort invested in Section 4.3, the performance in Section 4.4 and the limitations of Jet in Section 4.5. We conclude the discussion with a comparison to a project with similar goals named Pig in Section 4.6.

4.1 Optimizations

Jet only applies compile time optimizations. This means that a wide range of important optimizations are currently not considered [4]. The optimizer we use is not based on a cost model, therefore we only consider optimizations which do not hinder the performance in edge cases.

Within these constraints however we found generally applicable implementations and implemented them. In Figure 4.1 we show the combined effect of our optimizations for the parsing and filtering of *lineitems* in TPCH Query 12 [12]. In the unoptimized code, 16 fields are parsed and transmitted over the network. In the narrowed code, which only considers the fields that are used by the program, only 5 fields are parsed and only one is sent across the network. In the code with all our optimizations enabled, the parsing and the filtering becomes interleaved. The fully optimized code therefore only looks at the minimal fields necessary for discarding one item.

Our Projection Insertion algorithm works for nested objects and over all program constructs. This is due to the reuse of the extensive facilities in LMS. To implement Projection Insertion for Jet, it was enough to perform the analysis, insert narrowers before barriers, and define suitable types.



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

```
def x776(x10: (java.lang.String)) = {
  val x12 = x11.split(x10, 16);
  val x13 = x12(0);
  val x14 = x13.toInt;
  val x33 = x12(10);
  val x34 = ch.epfl.distributed.datastruct.Date(x33);
  val x35 = x12(11);
  val x36 = ch.epfl.distributed.datastruct.Date(x35);
  val x37 = x12(12);
  val x38 = ch.epfl.distributed.datastruct.Date(x37);
  val x40 = x12(14);
  val x296 = new LineItem_0_10_11_12_14(x14, x34, x36, x38, x40);
  x296: LineItem
}
val x777 = x9.map(x776);
def x853(x67: (LineItem)) = {
  val x68 = x67.l_shipmode;
  val x69 = x68 == x6;
  val x71 = if (x69) {
   true
  } else {
   val x70 = x68 == x7;
   x70
  }
  x71: Boolean
}
val x854 = x777.filter(x853);
def x855(x74: (LineItem)) = {
  val x75 = x74.l_receiptdate;
  val x76 = x5 <= x75;
  x76: Boolean
}
val x856 = x854.filter(x855);
def x857(x79: (LineItem)) = {
  val x80 = x79.l_shipdate;
  val x81 = x79.l_commitdate;
  val x82 = x80 < x81;
  x82: Boolean
}
val x858 = x856.filter(x857);
def x859(x85: (LineItem)) = {
  val x86 = x85.l_commitdate;
  val x87 = x85.l_receiptdate;
  val x88 = x86 < x87;
  x88: Boolean
}
val x860 = x858.filter(x859);
def x861(x91: (LineItem)) = {
  val x92 = x91.l_receiptdate;
  val x94 = x92 < x93;
  x94: Boolean
}
val x862 = x860.filter(x861);
def x852(x102: (LineItem)) = {
  val x103 = x102.l_orderkey;
  val x278 = x102.l_shipmode;
  val x279 = new LineItem_14(x278);
  val x280 = (x103, x279);
  x280: scala.Tuple2[Int, LineItem]
}
val x863 = x862.map(x852);
```

```
val x3849 = x9.flatMap(input => {
  val out = scala.collection.mutable.Buffer[scala.Tuple2[Int, LineItem]]();
  {
   val x920 = input // loop var x918;
   val x1429 = x11.split(x920, 16);
   val x1438 = x1429(14);
   val x3303 = x1438 == x6;
   val x3305 = if (x3303) {
    true
   } else {
    val x3304 = x1438 == x7;
    x3304
   }
   val x3847 = if (x3305) {
    val x1436 = x1429(12);
    val x1437 = ch.epfl.distributed.datastruct.Date(x1436);
    val x3608 = x5 <= x1437;
    val x3845 = if (x3608) {
     val x1432 = x1429(10);
     val x1433 = ch.epfl.distributed.datastruct.Date(x1432);
     val x1434 = x1429(11);
     val x1435 = ch.epfl.distributed.datastruct.Date(x1434);
     val x3609 = x1433 < x1435;
     val x3843 = if (x3609) {
      val x3794 = x1435 < x1437;
      val x3806 = if (x3794) {
       val x3795 = x1437 < x93;
       val x3802 = if (x3795) {
        val x1430 = x1429(0);
        val x1431 = x1430.toInt;
        val x3796 = new LineItem_14(x1438);
        val x3797 = (x1431, x3796);
        out += x3797 // yield
       }
      }
     }
    }
   }
  }
  out
})
```

(a) Unoptimized **(b) Narrowed** **(c) Fully optimized**

Figure 4.1: Generated code snippet for TPCH Q12 lineitem mapper

4.2 Extensibility

We claim that Jet is extensible. We relied on the extensibility twice during the implementation of Jet, and both time it met our needs.

In the first case, we were not getting the performance we were hoping for from the Scoobi framework. Since the code generation layer is very thin and completely separate from the rest of Jet, we were able to add support for Crunch within one week. This includes choosing Crunch, getting to know it, adding a new serialization scheme and implementing the code generation.

The second time we used the extensibility to add a multi-dimensional point module (Vector) for use in the k-means benchmark presented in Section 4.4.3. We created a Vector implementation in just about 100 lines of code, which features a high level interface but generates code that uses no high level constructs, just arrays and while loops.

We also used the extensibility to optimize regular expressions. Both the Vector and the regular expression optimizations can be reused in other LMS based DSL's.

4.3 Effort

We reused LMS and FlumeJava implementations. This allowed us to focus our efforts on the distributed data parallel computing domain. The effort invested specifically into Jet amounts to just 5 Person months. This includes getting to know LMS, finding suitable optimizations and performing benchmarks. As of this writing, the main code for Jet (helper classes for the generated code excluded) amounts to around 3700 lines of Scala.

4.4 Performance

We performed experiments to test our optimizations in a realistic setting. We used the Amazon EC2 cloud and rented 21 m1.large machines, 20 slaves and 1 master, in the same availability zone for these test runs. They each have 2 CPU cores, 7.5 Gb Ram, 2 hard disks and 1 GBit Ethernet connections. We ran each benchmark at least 3 times, and took the average. The observed standard deviations were smaller than 3%, so we omitted them from the graphs.

For the Hadoop based tests, we used Cloudera's Hadoop distribution cdh3u4 [10]. We started a cluster using Whirr 0.7.1 [7], and made sure that both physical hard drives were used for the distributed filesystem. We did not tweak any settings for Hadoop. We used Crunch version 0.2.4 and Scoobi version 0.4.0, the newest stable releases. For serialization of user classes, we selected our generated `Writable` (the interface for serialization of user classes in Hadoop) implementations, as they seemed to perform faster than the other alternatives.

For the Spark based tests, we used the Mesos [14] EC2 scripts to start a cluster. We had to tweak several settings in order to ensure successful program runs: We set the default parallelism to the number of cores in the cluster, and increased the available memory for Spark slaves to 6 Gb. For serialization, we used Kryo [2], the standard for Spark.

4.4.1 WordCount

The WordCount benchmark counts the appearances of all words in Wikipedia articles. The input is a dump of Wikipedia articles, that contains a plaintext version that was generated from an XML version. This extraction did not consistently remove all of the markup, so we used five regular expressions to remove the incorrectly parsed parts from the output, all on the mapper side. This job consists of one *map* and one *reduce* phase, and the former is much more expensive than the latter. This makes for a good benchmark of

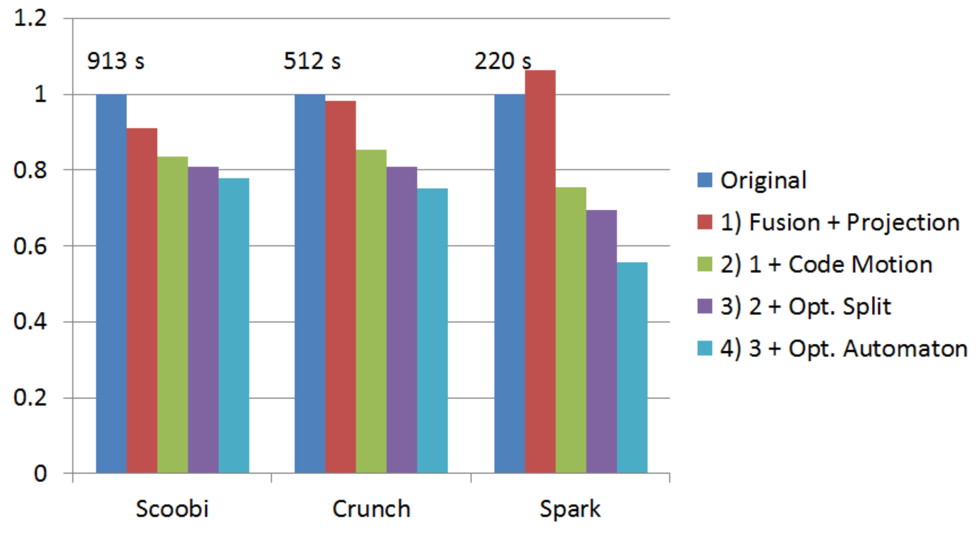


Figure 4.2: WordCount benchmark

a frameworks base performance in the mapping stage, while still giving us opportunities to test the optimizations for regular expressions. Projection Insertion can not reduce the network traffic for this program.

We tested our program on a 62 Gb plaintext input, the articles file of the AWS dataset [5] copied twice. It is a tab separated file, and we only used the last column, the one with the plaintext content of the article.

We present the results of our test runs in Figure 4.2. The y-axis is normalized to the unoptimized version for each framework. The number above the unoptimized column is the total job time for that framework. For this graph, we added the optimizations one by one. We start with the naive unoptimized version, then we add first Loop Fusion and Projection Insertion, which can not do much for this benchmark. Only on Scoobi we see a 10% speedup. Then we add Code Motion, which in this case means that the regular expressions are only calculated once and then reused. This reduces the total job time by 65–70 seconds across all frameworks. The next version adds the optimized splitter, which gives us around 13 seconds on Spark, but around 23 seconds on the Hadoop based frameworks. The next version has also the use of the faster automaton library enabled, which reduces the total job time further by 26–30 seconds on all frameworks. Our optimizations combined yield a speedup of 28% on Scoobi, 33% on Crunch and 79% on Spark.

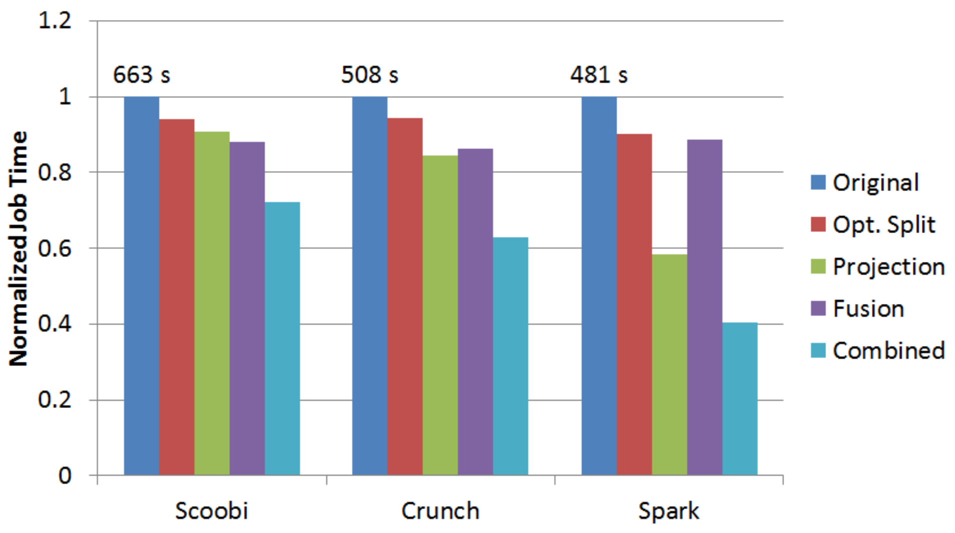


Figure 4.3: TPCH Q12 benchmark

4.4.2 TPCH Q12

TPCH Q12 reads two collections, joins them, and then combines all values into two values. This needs two Hadoop jobs, and the data involved in the first one is large. Since after the join only one field of each collection is used, Projection Insertion can remove most of the fields here for this job.

We chose this program to test the Projection Insertion, but to also show the effects of Loop Fusion and the optimized splitter. As input we used a generated dataset, created with the program *dbgen* by the TPC Performance Council [12] with a scaling factor of 100. The total input was around 100 GB of plaintext.

We present the obtained results in Figure 4.3. The graph follows the same conventions as Figure 4.2, except that here the optimizations are added individually to the unoptimized versions. The last column shows the performance of code with all optimizations enabled.

In this program all frameworks have a similar base performance. The optimized splitter reduces the job time by 28 seconds on Crunch and by 47 seconds on Spark. Projection Insertion results only in small speedups (10% Scoobi / 19% Crunch) for Hadoop, while Spark exhibits a speedup of 71%. Fusion can help by reducing the number of fields that are parsed. This improves performance by 13–16% on all frameworks. The fully optimized program gives us a speedup of 38% on Scoobi, 59% on Crunch and 148% on Spark.

We see that the Projection Insertion has a great impact on Spark. We believe

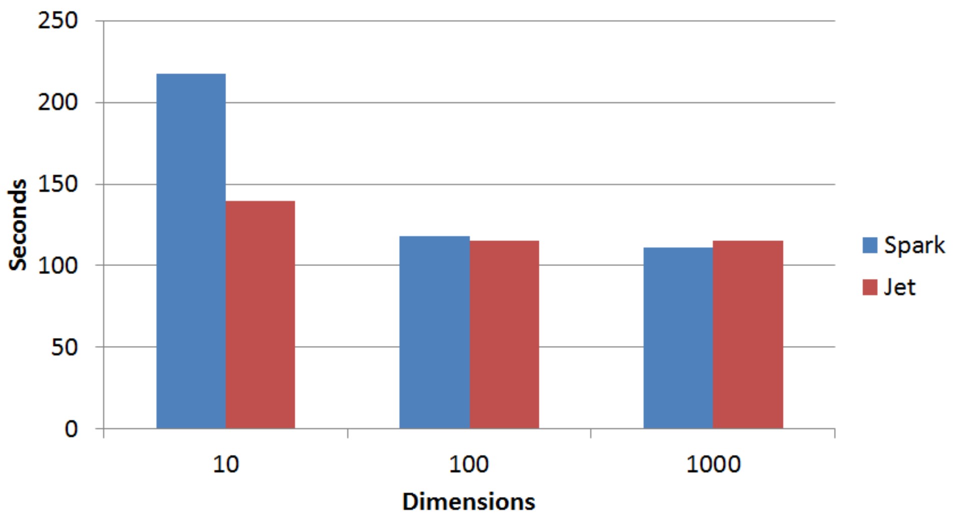


Figure 4.4: K-means

this to be due to some performance problems with spilling caches in Spark.

4.4.3 K-means

K-means is an iterative clustering algorithm. It reads the same data set in each iteration to update the cluster centers. As shown by Zaharia et al. [29], this job runs between 2x and 3x faster on Spark than on Hadoop, due to Spark's caching abilities. We have therefore only tested this program on Spark.

We ported the implementation from the Spark repository [1] to Jet. As mentioned in Section 4.2, we created a DSL module for multi-dimensional points and updated the implementation to use it instead of the wrapper around an array that was in the original. We then compared the performance of these two versions.

K-means can not profit from Projection Insertion, because no data can be eliminated. Similarly it can not profit from Loop Fusion, since the way this program has been written, this was already done by the developer. The only optimization we are testing in this case is therefore our added module.

We generated input data for this test, each around 20 Gb, with 10 - 1000 dimensions. We set a fixed number of iterations and used 50 centers. The total amount of computation is therefore comparable for different dimensions.

We present the results for this benchmark in Figure 4.4. The y-axis is the total job time in seconds, the x-axis is the number of dimensions. The performance between the two version only differs substantially for the case with 10 dimensions, where our version is 36% faster. We believe that the overhead imposed by the iterator is noticeable for 10 iterations, but is completely removed with 1000 dimensions. Our results are similar to those presented by Murray et al. [20].

4.5 Limitations

Due to the prototype nature of the current implementation there are some limitations, which possibly can be overcome:

- Jet currently only allows simple struct types that do not allow any form of inheritance. This is a complicated problem, that will require some further investigation.
- LMS only provides a limited number of methods compared to regular Scala. There are plans for enabling simple method calls to regular Scala code, but in the meantime it is possible to add individual calls as needed.

4.6 Jet compared to Pig

Pig [24] is another DSL for Distributed Data Parallel Computing. It has similar goals and it also optimizes regular expressions and performs Projection Insertion. For this reason we compare the performance of Jet's generated code with Pig.

We implemented the WordCount and TPCH Q12 benchmarks also for Pig. For TPCH Q12, the code size for the Pig version was comparable to the implementation in Jet, for WordCount, we needed a User Defined Function (UDF) in Pig to perform the custom split. The code for the WordCount in Pig is therefore substantially larger than the one in Jet. Also it required two programming languages: Pig Latin for the main program logic and another one (we used Java) for the split.

In Figure 4.5 we show the performance of the programs in Pig and the versions of Jet with all optimizations enabled. The numbers for Pig and Crunch are directly comparable. Both are as optimized as possible by that framework, and both run on Hadoop. The numbers for Spark are not directly comparable to the ones for Hadoop, because there are some static overheads for Hadoop. For example, the Hadoop programs all perform the FlumeJava optimizations at runtime, create a jar of the program, and send it to the job tracker which then distributes the jar on the cluster. In Spark on the other hand, the jar file has to be already present on all computers.

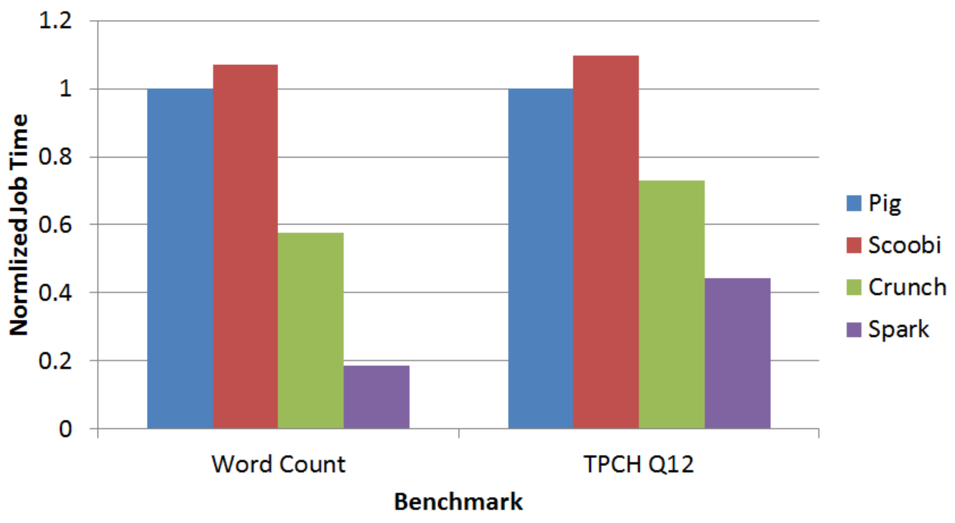


Figure 4.5: Pig vs Jet

	Pig	Crunch Original	Crunch Optimized
WordCount	664.7	511.7 (-23%)	384.3 (-42%)
TPCH Q12	437.2	508.1 (+16%)	319.2 (-27%)

Table 4.1: Pig vs Jet total job times in seconds

We see that our Crunch versions outperform Pig by 42% for WordCount and by 27% for TPCH Q12. We also see that Scoobi performs consistently worse than Crunch and Pig.

We present the obtained numbers for Pig, the unoptimized and version with all optimizations enabled as generated by Jet for Crunch in Table 4.1. The percentages in the parentheses show the speedup vs Pig. Notice that the unoptimized version for the WordCount outperforms Pig, this could be due to inefficient `String` operators in Pig. For TPCH Q12 however we need to apply our optimizations to be faster than Pig.

Chapter 5

Related and Future Work

We survey the work on optimizing distributed data parallel programs. We focus on work implementing optimizations at compile time and exclude work optimizing the runtime itself, as it is complementary to our work. After the survey of related work, we will discuss how Jet can be extended.

5.1 Pig

Pig [24] is a framework for Hadoop, allowing to express programs in a domain specific language called Pig Latin. Pig Latin tries to find a sweet spot between the declarative SQL and the low-level, procedural style of MapReduce. Pig Latin is a very restricted language, offering no functions or loops, and comes with its own type system. The corresponding runtime Pig applies relational optimizations that include early projection, pushing down of filters and operator rewrites.

Pig is extended by specifying User Defined Functions (UDF) in another general purpose programming language. Pig will not analyze those functions, so using a UDF will lead to missed opportunities for optimizations. To make the programming of iterative jobs possible, the Pig Latin script has to be embedded in another language and repeatedly compiled and called from there.

Pig supports many common relational operators like sorting, removing duplicate values, counting, and different kind of joins. For the inner join it provides multiple implementations with different performance characteristics.

Jet and Pig try to achieve similar goals, but in very different ways. Pig was implemented from the ground up, only reusing Hadoop for the runtime itself. It uses general purpose implementations for the relational operators, that are configured and chained together at runtime. Jet on the other hand reuses existing FlumeJava implementations and LMS, and uses code generation to specialize the operators for each program. Reusing other projects has allowed us to focus our efforts on domain specific problems, getting programming language semantics, parser and a type system for free. Pig is a mature project and currently supports more relational operators than Jet does, but we believe we could implement those missing operators with a fraction of the effort that was required for Pig.

Pig defines its own language Pig Latin. This not only means that they must implement parser, semantics and type system for this language, but that the users also must learn it. The tool support has also been developed, but that too required extensive efforts. On the other hand, Jet is deeply embedded in Scala and tool support for Scala programs can be used.

5.2 Hive

Hive [27] is a complete warehousing solution based on Hadoop. Like Pig, Hive defines its own language called HiveQL, which closely resembles SQL. Hive does not only have primitive types to define the columns, but also features structs, maps and lists. Hive too is extensible by user defined functions.

Hive performs many of the same optimizations as Pig, the rule based rewrites include Projection Insertion and executing filters as early as possible. Hive, like Pig, allows the user to choose a suitable join implementation, including map side joins.

In contrast to Pig and Jet, Hive also manages the metadata for tables centrally, so that the queries can simply refer to tables instead of paths in HDFS. It also allows partitioning a table by creating subfolders, selections on these partitions can then be used to preselect which folders to read.

To compare Hive with Jet, the same points can be made as for the comparison between Pig and Jet. While HiveQL resembles SQL even more so than Pig Latin does, a considerable effort had to go into defining the parser and semantics of the language. Hive does not support loops.

5.3 Scope

Scope [8] features also a declarative language inspired by SQL. It runs on the Cosmos system, a Microsoft internal system, which allows a much finer grained control over the execution of a distributed program than Spark or Hadoop do. In Cosmos, arbitrary code and resources can be uploaded to a machine and then executed there. Scope employs a cost based optimizer to use the resources efficiently and create a complete execution plan. Scope performs relational operations too, like Projection Insertion, early filtering and pre-aggregation. Like Pig, Scope can be extended by user defined functions.

Further building upon Scope, RoPE [4] has been presented, an instrumentation based analysis and optimization for Scope programs. RoPE gathers various statistics about the data properties and information about the memory and processing cost of functions, and uses these to configure the cost estimates for the Scope optimizer. Even when the job can not be changed anymore, jobs are often frequently recurring, and the gained knowledge can be used when starting the job the next time.

Jet currently does not generate such a detailed execution plan, therefore some of the optimizations done in Scope are not possible. While Jet is embedded in Scala and does not need a second language, Scope is like Pig split into the declarative language and the general purpose language for user defined functions. While in Jet one can define rows as normal Scala case classes, in Scope they have to be defined as rows, and these definitions need to exist in both the Scope code and the C# code for a user defined function.

Like Pig, Scope does not analyze the user defined functions themselves, it just treats them as black boxes. But unlike with Pig and Hive, this does not matter as much, as the properties of the function are collected at runtime by instrumentation. Instrumentation could be done in Jet, but currently we do not have a cost based optimizer. Additionally, as long as we target Hadoop or Spark, our freedom for rewriting the program is more limited.

5.4 Hadoop program optimizers

Manimal [17] and HadoopToSQL [16] optimize a Hadoop program by rewriting the byte code. They analyze one Hadoop job to infer properties about it, which they use to insert relational optimizations, for example optimizing the data access patterns by building indexes. Manimal can further rewrite the program to use columnar storage instead. Being based on byte code analysis, both projects suffer from lost information due to compilation.

Jet is used as a preprocessor, not a post processor. This means more high level information is available, but unlike these projects, the user has to start his program using Jet. Currently Jet does not do any index based optimization, but we believe that we could achieve this with less effort than Manimal and HadoopToSQL, and for a larger set of programs due to our high-level information about the program.

5.5 Future Work

Jet can be extended in various ways. We are currently only implementing one relational optimization, so other optimizations like early filtering could be added. But also column storage and indexing optimizations would be interesting and within the possibilities.

Also, currently all optimizations are applied at compile time, but staging could just as well be done at runtime. At runtime, more data is available like the size of input files, this would enable us to reorder joins for example. A natural extension of staging at runtime would then be to instrument and reoptimize the code as it has been done for Scope [4].

Further we could also combine Jet with other LMS based DSLs. Regular expressions and XML processing could benefit from staged code generation. LMS can also generate code for GPUs, we could also try to leverage GPUs for computationally intensive distributed computations.

Chapter 6

Conclusion

We present Jet, a DSL for distributed data parallel computing that has a high level declarative programming model. Through lightweight modular staging Jet is deeply embedded in Scala, allowing to reuse Scala's type system and parser. Jet is modular and extensible and allows execution on both Spark and Hadoop. Jet optimizes programs by employing classical compiler optimizations like Code Motion and Loop Fusion, and relational optimizations like Projection Insertion. The presented optimizations accelerate the tested programs between 79–148% on Spark and 33–59% on Hadoop.

The overall lesson that we draw from this work is that an extensible and modular compiler is a good approach for writing a domain specific language. Modularity allows the reuse of components between DSLs, so that we could focus completely on domain specific issues. Extensibility allows Jet to be useful even if it does not support a certain use case, because the programmer can just add his own module and optimizations for it. We hope to see more extensible and modular compilers in the future.

Appendix A

Serialization

We generated code for the serialization of user classes automatically to enable Projection Insertion. They fulfill the following properties:

- Only the live fields are serialized
- Only the live fields are needed to call the constructor
- Simple code is generated, so that efficient accessors can be used

We generate these types differently for each target, to satisfy additional constraints imposed by it.

A.1 Hadoop

For Hadoop, we generate implementations for the `Writable` interface. A class implementing `Writable` needs a constructor without arguments, and needs to define how to read fields from a stream and how to write fields to a stream.

To allow any subsets of fields to be used, we defined two constructors: One without any arguments and one with an argument for each field, and each argument has a default value. Scala allows us to invoke the constructor with any combination of fields by naming them.

To define which fields should be serialized, we use a bitset. If the bit at the position i is 1, that field is serialized. When serializing an object, we first write the bitset and then the fields. Similarly when deserializing an object, we first read the bitset and then the fields. For each combination of fields that appear in the program we specialize the reading and writing methods.

Listing 7 shows generated code for the `Person` class, Listing 8 shows a corresponding valid constructor invocation. Note that the bitset is just a field, if it were a constructor argument it would incorrectly appear in the case classes String representation.

A.2 Spark

For Spark, the requirements are different. Here we are not restricted to a single class, but we need to ensure that the memory usage is as minimal as possible of the generated objects.

We generate a trait with all fields as methods, that just throw an exception when accessed. Then we create subclasses for this trait, with different combinations of fields. The naming convention simply concatenates the field numbers that are available in that type.

Listing 9 shows generated code for the `Person` class. Listing 10 shows a corresponding valid constructor invocation.

The serialization is in this case not generated by Jet but will be generated at runtime by the Kryo serialization framework [2].

```
case class Person(var name: java.lang.String = " ",
        var age: Int = 0, var zipcode: Int = 0,
        var email: java.lang.String = " ")
     extends Writable {
  def this() = this(name = " ")

  var __bitset: Long = 15

  override def readFields(in: DataInput) {
    __bitset = WritableUtils.readVLong(in)
    __bitset match {
      case 15 => readFields_0_1_2_3(in)
      case 3 => readFields_0_1(in)
      case x => throw new RuntimeException(
        "Unsupported bit combination " + x)
    }
  }
  override def write(out: DataOutput) {
    WritableUtils.writeVLong(out, __bitset)
    __bitset match {
      case 15 => write_0_1_2_3(out)
      case 3 => write_0_1(out)
      case x => throw new RuntimeException(
        "Unsupported bit combination " + x)
    }
  }
  def readFields_0_1_2_3(in: DataInput) {
    name = in.readUTF
    age = WritableUtils.readVInt(in)
    zipcode = WritableUtils.readVInt(in)
    email = in.readUTF
  }
  def write_0_1_2_3(out: DataOutput) {
    out.writeUTF(name)
    WritableUtils.writeVInt(out, age)
    WritableUtils.writeVInt(out, zipcode)
    out.writeUTF(email)
  }
  def readFields_0_1(in: DataInput) {
    name = in.readUTF
    age = WritableUtils.readVInt(in)
  }
  def write_0_1(out: DataOutput) {
    out.writeUTF(name)
    WritableUtils.writeVInt(out, age)
  }
}
```

Listing 7: `Writable` implementation for Hadoop

```
val person = new Person(name="Mickey Mouse", age=84)
person.__bitset = 3
```

Listing 8: Instantation of `Person` with only two fields with the generated `Writable`

```
trait Person extends Serializable {
  def throwException(name: String) =
        throw new RuntimeException(
                "Should not try to access "+name+" here,"+
                " internal error")

  def name: java.lang.String = throwException("name")
  def age: Int = throwException("age")
  def zipcode: Int = throwException("zipcode")
  def email: java.lang.String = throwException("email")
}

case class Person_0_1(
        override val name: java.lang.String,
        override val age: Int)
        extends Person

case class Person_0_1_2_3(
        override val name: java.lang.String,
        override val age: Int,
        override val zipcode: Int,
        override val email: java.lang.String)
        extends Person
```

Listing 9: Generated classes for Spark

```
val person = new Person_0_1("Mickey Mouse", 84)
```

Listing 10: Instantiation of `Person` with only two fields with the generated classes for Spark

Bibliography

[1] Spark k-means implementation. https://github.com/mesos/spark/blob/master/examples/src/main/scala/spark/examples/SparkKMeans.scala.

[2] Kryo: Fast, efficient java serialization and cloning. http://code.google.com/p/kryo/, 2012.

[3] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded dsl for high performance big data processing. To appear in BigData, 2012. https://github.com/vjovanov/bigdata2012/raw/draft/build/paper.pdf.

[4] S. Agarwal, S. Kandula, N. Bruno, M. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, page 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[5] Amazon. Public data set catalog: Wikipedia extraction (wex). http://aws.amazon.com/datasets/2345, 2009.

[6] Apache. Hadoop: Open-source software for reliable, scalable, distributed computing. http://hadoop.apache.org, 2012.

[7] Apache. Whirr: A set of libraries for running cloud services. http://whirr.apache.org, 2012.

[8] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. In *In International Conference of Very Large Data Bases (VLDB*, 2008.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, page 363–375, New York, NY, USA, 2010. ACM.

[10] Cloudera. Cdh: Open source, enterprise-ready hadoop distribution. https://ccp.cloudera.com/display/SUPPORT/CDH+Downloads.

[11] Cloudera. Crunch: Simple and efficient java library for mapreduce pipelines. https://github.com/cloudera/crunch, 2012.

[12] T. P. Council. Tpc benchmark^{TM}. http://www.tpc.org/tpch/.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, page 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[15] 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.

[16] 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.

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

[18] A. Møller. dk. brics. automaton–finite-state automata and regular expressions for java, 2005. http://www.brics.dk/automaton/.

[19] 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.

[20] 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*, page 121–131, 2011.

[21] NICTA. Scoobi: A scala productivity framework for hadoop. http://nicta.github.com/scoobi/, 2012.

[22] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification*. Citeseer, 2004.

[23] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, page 41–57, New York, NY, USA, 2005. ACM.

[24] 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.

[25] 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.

[26] J. M. Smith and P. Y. Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, Oct. 1975.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. *Data Engineering, International Conference on*, 0:996–1005, 2010.

[28] 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.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, page 2–2, Berkeley, CA, USA, 2012. USENIX Association.