# Stratosphere

Amir H. Payberah
Swedish Institute of Computer Science

amir@sics.se

April 22, 2014

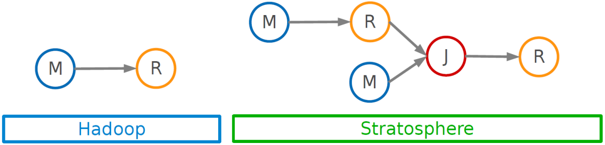
# Motivation

- MapReduce programming model has not been designed for complex operations, e.g., data mining.

- Very expensive, i.e., always goes to disk and HDFS.

## Proposed Solution

- Extends MapReduce with more operators.
- Support for advanced data flow graphs.



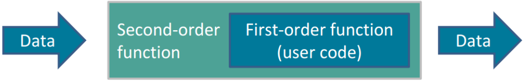
# Stratosphere Programming Model (PACT)

# Stratosphere Programming Model (1/2)

- Stratosphere's programming model is based on parallelizable operators.

- Parallelizable operators are higer-order functions that execute user-defined (first-order) functions in parallel on the input data.

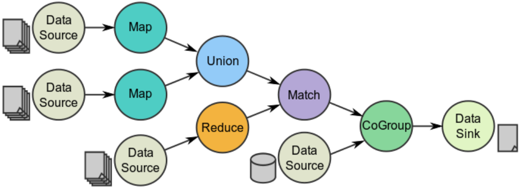
- They are also called transformation or second-order functions.



# Stratosphere Programming Model (2/2)

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.

- Job description based on directed acyclic graphs (DAG).

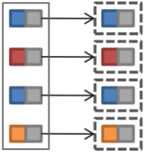


# Transformations (1/7)

- Map
- Reduce
- Join
- Cross
- CoGroup
- Union

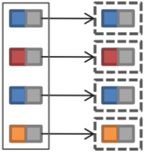
# Transformations: Map (2/7)

- All pairs are independently processed.

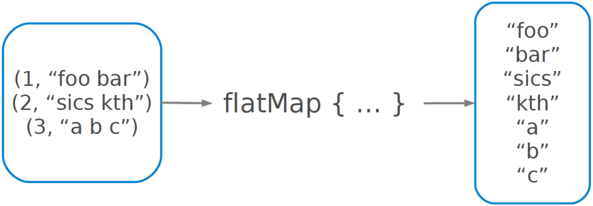


# Transformations: Map (2/7)

- All pairs are independently processed.



```
val input: DataSet[(Int, String)] = ...
val mapped = input.map { (a, b) => (a + 2, b) }
val filtered = input.filter { (a, b) => a > 3 }
val mapped2 = input.flatMap { _._2.split(" ") }
```



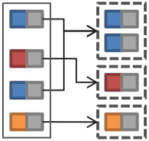
(1, “foo bar”)
(2, “sics kth”)
(3, “a b c”)

flatMap { ... }

“foo”
“bar”
“sics”
“kth”
“a”
“b”
“c”

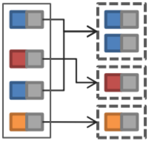
# Transformations: Reduce (3/7)

- Pairs with identical key are grouped.
- Groups are independently processed.

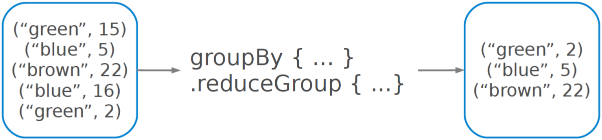


# Transformations: Reduce (3/7)

- Pairs with identical key are grouped.
- Groups are independently processed.



```
val input: DataSet[(String, Int)] = ...
val reduced = input
  .groupBy { _._1 }
  .reduceGroup { _.minBy {_._2} }
```



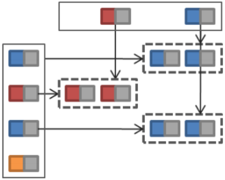
("green", 15)
("blue", 5)
("brown", 22)
("blue", 16)
("green", 2)

groupBy { ... }
.reduceGroup { ...}

("green", 2)
("blue", 5)
("brown", 22)

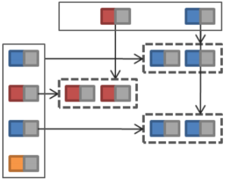
# Transformations: Join (4/7)

- Performs an equi-join on the key.
- Join candidates are independently processed.

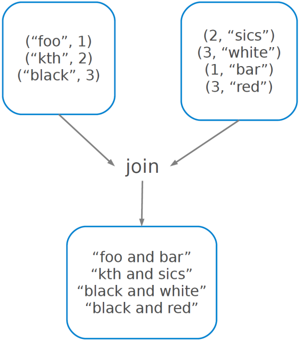


# Transformations: Join (4/7)

- Performs an equi-join on the key.
- Join candidates are independently processed.

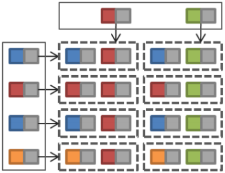


```
val counts: DataSet[(String, Int)] = ...
val names: DataSet[(Int, String)] = ...
val join = counts.join(names)
  .where { _._2 }
  .isEqualTo { _._1 }
  .map { (l, r) => l._1 + "and" + r._2 }
```



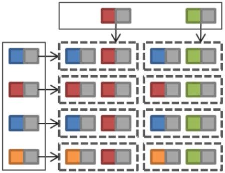
# Transformations: Cross (5/7)

- Builds a Cartesian Product (CP).
- Elements of CP are independently processed.



# Transformations: Cross (5/7)

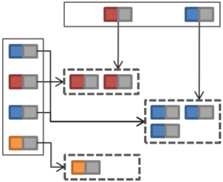
- Builds a Cartesian Product (CP).
- Elements of CP are independently processed.



```
val left: DataSet[(String, Int)] = ...
val right: DataSet[(String, Int)] = ...
val crossed = left.cross(right)
  .map { (l, r) => ... }

val crossed2 = left.cross(right)
  .flatMap { (l, r) => ... }
```

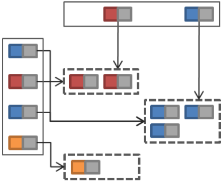
# Transformations: CoGroup (6/7)

- Groups each input on key.
- Groups with identical keys are processed together.



# Transformations: CoGroup (6/7)

- Groups each input on key.
- Groups with identical keys are processed together.



```
val counts: DataSet[(String, Int)] = ...
val names: DataSet[(Int, String)] = ...

val cogrouped = counts.cogroup(names)
  .where { _._2 } isEqualTo { _._1 }
  .map { (l, r) => ... }

val cogrouped2 = counts.cogroup(names)
  .where { (_, c) => c } isEqualTo { (n, _) => n }
  .map { (l, r) => ... }
```

# Transformations: Union (7/7)

- Union is an operator without a user-defined function.
- It merges two or more input data sets into a single output data set using bag semantics, i.e., duplicates are not removed.

# DataSet

- DataSet is the core of the Stratosphere Scala API.

- It looks and behaves like a regular Scala collection.

- It does not contain any actual data but only represents data.

- An operation on DataSet creates a new DataSet.

```
val input: DataSet[(String, Int)] = ...
val mapped = input.map { a => (a._1, a._2 + 1) }
```

# Skeleton of a Stratosphere Program

1. Data source: text file, JDBC, CSV, etc.
   - Loaded in internal representation: DataSet

2. Transformations on DataSet: map, reduce, join, etc.
   - Higher-order function

3. Data sink: text file, JDBC, CSV, etc.

# Data Source

```
// type: DataSet[String]
val input = TextFile("hdfs://")

// type: DataSet[(Int, String)]
val input = DataSource("file://", CsvInputFormat[(Int, String)]())

// type: DataSet[(Int, Int)]
def parseInput(line: String): (Int, Int) = {...}
val input = DataSource("hdfs://", DelimitedInputFormat(parseInput))
```

# Data Sink

```
val counts: DataSet[(String, Int)] = ...

val sink = counts.write("hdfs://", CsvOutputFormat())

def formatOutput(a: (String, Int)): String = {
  "Word " + a._1 + " count " + a._2
}
val sink = counts.write("file://", DelimitedOutputFormat(formatOutput))
```

# Example: Word Count

```
val input = TextFile(textInput)

val words = input.flatMap { _.split(" ") map { (_, 1) } }

val counts = words.groupBy { case (word, _) => word }
  .reduce { (w1, w2) => (w1._1, w1._2 + w2._2) }

val output = counts.write(wordsOutput, CsvOutputFormat())
```

# Example: Word Count

Data source

```
val input = TextFile(textInput)

val words = input.flatMap { _.split(" ") map { (_, 1) } }

val counts = words.groupBy { case (word, _) => word }
  .reduce { (w1, w2) => (w1._1, w1._2 + w2._2) }

val output = counts.write(wordsOutput, CsvOutputFormat())
```

Transformation

Data sink

# Example: Word Count - Local Execution

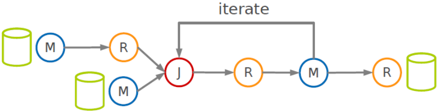
```
val input = TextFile(textInput)

val words = input.flatMap { _.split(" ") map { (_, 1) } }

val counts = words.groupBy { case (word, _) => word }
  .reduce { (w1, w2) => (w1._1, w1._2 + w2._2) }

val output = counts.write(wordsOutput, CsvOutputFormat())

val plan = new ScalaPlan(Seq(output))

val ex = new LocalExecutor()
ex.start()
ex.executePlan(plan)
ex.stop()
```

# Example: Word Count - Remote Execution

```
val input = TextFile(textInput)

val words = input.flatMap { _.split(" ") map { (_, 1) } }

val counts = words.groupBy { case (word, _) => word }
  .reduce { (w1, w2) => (w1._1, w1._2 + w2._2) }

val output = counts.write(wordsOutput, CsvOutputFormat())

val plan = new ScalaPlan(Seq(output))

val ex = new RemoteExecutor("localhost", 6123, "target/some.jar")
ex.executePlan(plan)
```

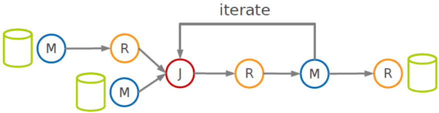
# Iteration (1/6)

- Loop over the working data multiple times.

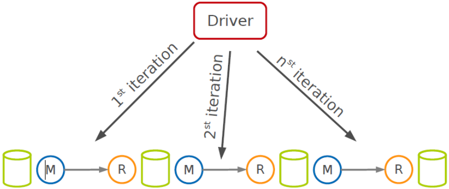


# Iteration (1/6)

- Loop over the working data multiple times.



- Iterations with hadoop
  - Slow: using HDFS
  - Everything has to be read over and over again



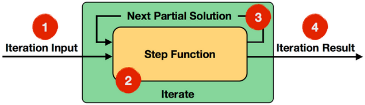
# Iteration (2/6)

- Two types of iteration at stratosphere:
  - Bulk iteration
  - Delta iteration

- Both operators repeatedly invoke the step function on the current iteration state until a certain termination condition is reached.

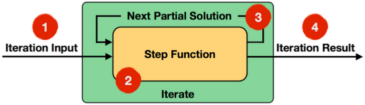
# Iteration - Bulk Iteration (3/6)

- In each iteration, the step function consumes the entire input (the result of the previous iteration, or the initial data set), and computes the next version of the partial solution.
- A new version of the entire model in each iteration.



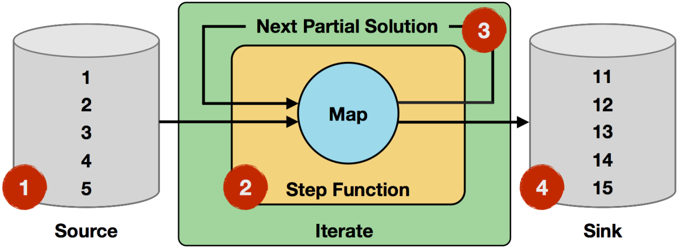
# Iteration - Bulk Iteration (3/6)

- In each iteration, the step function consumes the entire input (the result of the previous iteration, or the initial data set), and computes the next version of the partial solution.
- A new version of the entire model in each iteration.



```
val input: DataSet[Int] = ...

def step(partial: DataSet[Int]) = {
  val nextPartial = partial.map { a => a + 1 }
  nextPartial
}

val numIter = 10;
val iter = input.iterate(numIter, step)
```

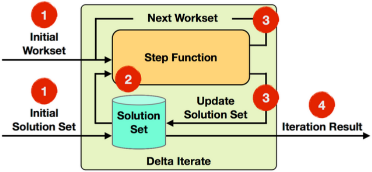
# Iteration - Bulk Iteration (4/6)



```
// 1st            2nd                     10th
map(1) -> 2       map(2) -> 3      ...    map(10) -> 11
map(2) -> 3       map(3) -> 4      ...    map(11) -> 12
map(3) -> 4       map(4) -> 5      ...    map(12) -> 13
map(4) -> 5       map(5) -> 6      ...    map(13) -> 14
map(5) -> 6       map(6) -> 7      ...    map(14) -> 15
```

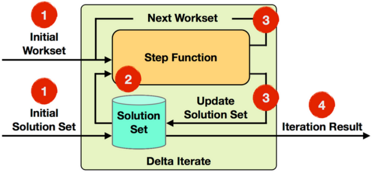
# Iteration - Delta Iteration (5/6)

- Only parts of the model change in each iteration.



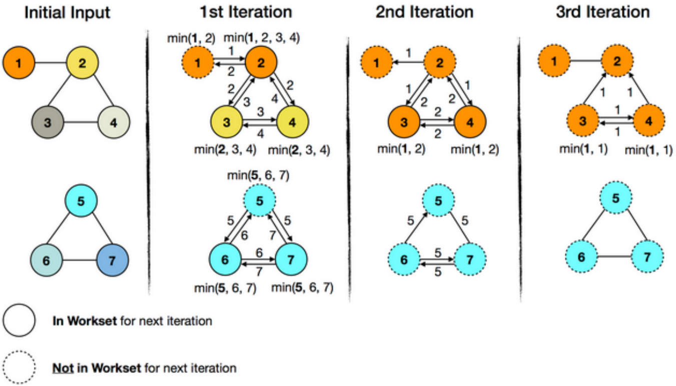
# Iteration - Delta Iteration (5/6)

- Only parts of the model change in each iteration.



```
val input: DataSet[(Int, Int)] = ...
val initWorkset: DataSet[(Int, Int)] = ...
val initSolutionSet: DataSet[(Int, Int)] = ...

def step(ss: DataSet[(Int, Int)], ws: DataSet[(Int, Int)]) = {
  val delta = ...
  val nextWorkset = ...
  (delta, nextWorkset)
}

val maxIter = 10;
val iter = input.iterateWithWorkset(initSolutionSet, initWorkset, step, maxIter)
```

# Iteration - Delta Iteration (6/6)



# Stratosphere Executin Engine (Nephele)

# Challenges

- Most of the existing processing frameworks, e.g., MapReduced, are designed for cluster environments.
    - Static and homogenous resources.
    - Not suitable for cloud environments.

- Given a set of compute resources, how to distribute the particular tasks of a job among them?

# Challenges

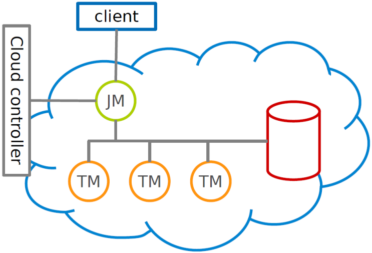
- Most of the existing processing frameworks, e.g., MapReduced, are designed for cluster environments.
  - Static and homogenous resources.
  - Not suitable for cloud environments.

- ~~Given a set of compute resources, how to distribute the particular tasks of a job among them?~~

- Given a job, what compute resources match the tasks the job consists of best?

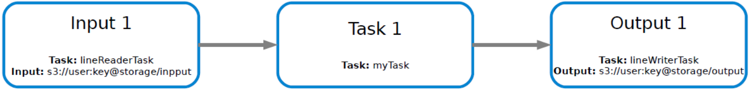
# Architecture

- Master-worker model
- Job Manager (JM): responsible for scheduling the received jobs and coordinating their execution.
- Task Manager (TM): receives tasks from the JM, executes them and informs the JM about their completion or possible errors.
    - Runs on VMs (instances)



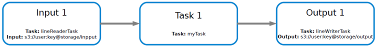
# Job Description

- Jobs are expressed as a directed acyclic graph (DAG), called job graph.
- Each vertex in the job graph representes a task.
- Users define tasks and their relations on an abstract level.
- They can also explicitly provide furthure annotations to their job.

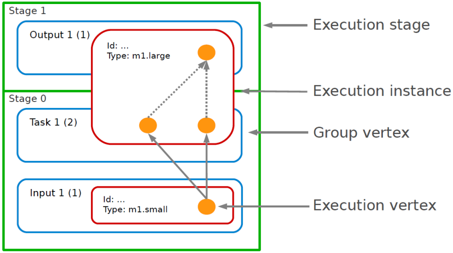


# Execution Graph

- Job graphs are transformed into the execution graph.
- Execution graph: informaiton to schedule and execute a job.
- Group vertex: corresponds to every vertex (task) of the job graph.
- Execution vertex: every task in a job graph is transformed into one or more subtasks (if the task is suitable for parallel execution).



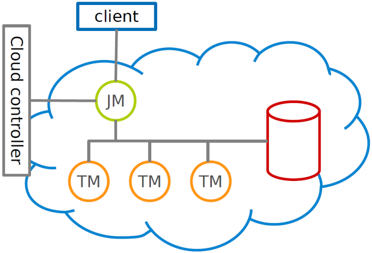
Job graph



Execution graph

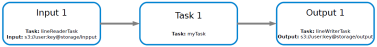
# Job Scheduling and Execution (1/2)

- Job graph is given to the Job Manager (JM).
- JM decides about the number of and type of instnaces (VM).
- The new instnaces boot up with a previously compiled VM image.
- The image starts a Task Manager (TM) and registers it with the JM.

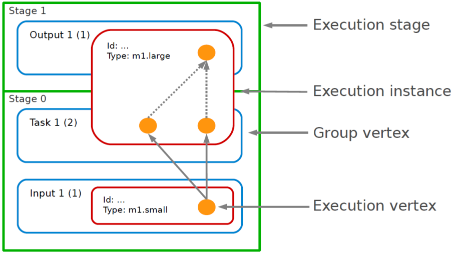


# Job Scheduling and Execution (2/2)

- The requested instances may not available: cause a problem.
- Separates the execution graph into execution stages.
  - Contains at least one group vertex.
  - Starts when all its preceding stages have been successfully processed.
  - When the processing of a stage begins, all instances required within the stage are allocated.



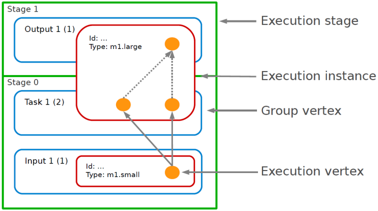
Job graph



Execution graph

# Channels (1/2)

- Network channels (pipeline)
  - Vertices must be in same stage.
- In-memory channels (pipeline)
  - Vertices must run on same VM.
  - Vertices must be in same stage.
- File channels
  - Vertices must run on same VM.
  - Vertices must be in different stage.



# Channels (2/2)

- Pipelined
  - Online transfer to receiver.
  - Receiver must be online.
  - Receiving speed limits sender speed.
- Materialized
  - Sender writes result to disk, and afterwards it transferred to receiver.
  - Materialized result can be used in a checkpoint for recovery.
  - Similar to Hadoop Map task results.

# Fault Tolerance

- Task failure compensated by backup task deployment.

- Lost intermediate results have to be reproduced
  - Track the execution graph back to the latest available result.
  - Latest available result may be input, if nothing is materialized.

- When a sender fails, an online receiver must be restarted.
  - If tasks are deterministic, the sender just disregards all input it has already seen.

# Summary

- PACT:
  - Extends MapReduce with more operations: map, reduce, join, cross, cogroup
  - Supports advanced data flow graph

- Nephele:
  - It is designed for cloud environments.
  - Transforms job graphs to execution graphs and executes its tasks over instances (VMs).

# Questions?

Acknowledgements

Some pictures were derived from the Stratosphere web site (`stratosphere.eu`).