The Stratosphere Big Data Analytics Platform

Amir H. Payberah
Swedish Institute of Computer Science

amir@sics.se
June 4, 2014
Big Data

small data

big data
Big Data refers to datasets and flows large enough that has outpaced our capability to store, process, analyze, and understand.
Where Does Big Data Come From?
The number of web pages indexed by Google, which were around one million in 1998, have exceeded one trillion in 2008, and its expansion is accelerated by appearance of the social networks.*

*“Mining big data: current status, and forecast to the future” [Wei Fan et al., 2013]
The amount of mobile data traffic is expected to grow to 10.8 Exabyte per month by 2016.*

*“Worldwide Big Data Technology and Services 2012-2015 Forecast” [Dan Vesset et al., 2013]
More than **65 billion devices** were connected to the Internet by 2010, and this number will go up to **230 billion** by 2020.*

*“The Internet of Things Is Coming” [John Mahoney et al., 2013]*
Many companies are moving towards using **Cloud services** to access **Big Data analytical tools**.
Open source communities
* Under development

** Stratosphere community is thinking to integrate with Mahout
Big Data looks tiny from Stratosphere
What is Stratosphere?

- An efficient distributed general-purpose data analysis platform.
- Built on top of HDFS and YARN.
- Focusing on ease of programming.
Research project started in 2009 by TU Berlin, HU Berlin, and HPI

An Apache Incubator project

25 contributors

v0.5 is released
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.
Solution

- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.
Stratosphere Programming Model
A program is expressed as an arbitrary data flow consisting of transformations, sources and sinks.
Higher-order functions that execute user-defined functions in parallel on the input data.
Higher-order functions that execute user-defined functions in parallel on the input data.
Transformations: Map

► All pairs are independently processed.
Transformations: **Map**

- All pairs are **independently** processed.

```scala
val input: DataSet[(Int, String)] = ...
val mapped = input.flatMap { case (value, words) => words.split(" ") }
```

Diagram:

```
(1, "foo bar")
(2, "sics kth")
(3, "a b c")
```

`flatMap` transformation: `foo` → `bar`, `sics` → `kth`, `a` → `b` → `c`
Transformations: Reduce

- Pairs with identical key are grouped.
- Groups are independently processed.
Transformations: Reduce

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

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

![Diagram](image-url)
Transformations: Join

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

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

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

val join = counts.join(names)
  .where(_._2)
  .isEqualTo(_._1)
  .map { case (l, r) => l._1 + "and" + r._2 }
```

```
("foo", 1)  
("kth", 2)  
("black", 3) 

(2, "sics")  
(3, "white")  
(3, "red")

join

"foo and bar"  
"kth and sics"  
"black and white"  
"black and red"
```
Groups each input on key
Transformations: and More ...

Groups each input on key

Builds a Cartesian Product
Transformations: and More ...

Groups each input on key

Builds a Cartesian Product

Merges two or more input data sets, keeping duplicates
Example: Word Count

```scala
val input = TextFile(textInput)

val words = input.flatMap { line => line.split(" ")
  map(word => (word, 1)) }

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

val output = counts.write(wordsOutput, CsvOutputFormat())
```
val large = env.readCsv(...) 
val medium = env.readCsv(...) 
val small = env.readCsv(...) 

joined1 = large.join(medium) 
  .where(_.3) 
  .isEqualTo(_.1) 
  .map { (left, right) => ... } 

joined2 = small.join(joined1) 
  .where(_.1) 
  .isEqualTo(_.2) 
  .map { (left, right) => ... } 

result = joined2.groupBy(_.3) 
  .reduceGroup(_.maxBy(_.2))
val large = env.readCsv(...)
val medium = env.readCsv(...)
val small = env.readCsv(...)

joined1 = large.join(medium)
    .where(_.3)
    .isEqualTo(_.1)
    .map { (left, right) => ... }

joined2 = small.join(joined1)
    .where(_.1)
    .isEqualTo(_.2)
    .map { (left, right) => ... }

result = joined2.groupBy(_.3)
    .reduceGroup(_.maxBy(_.2))
val large = env.readCsv(...)  
val medium = env.readCsv(...)  
val small = env.readCsv(...)  

joined1 = large.join(medium)  
    .where(_.3)  
    .isEqualTo(_.1)  
    .map { (left, right) => ... }  

joined2 = small.join(joined1)  
    .where(_.1)  
    .isEqualTo(_.2)  
    .map { (left, right) => ... }  

result = joined2.groupBy(_.3)  
    .reduceGroup(_.maxBy(_.2))
Join Optimization

```scala
val large = env.readCsv(...)
val medium = env.readCsv(...)
val small = env.readCsv(...)

joined1 = large.join(medium)
    .where(_.3)
    .isEqualTo(_.1)
    .map { (left, right) => ... }  

joined2 = small.join(joined1)
    .where(_.1)
    .isEqualTo(_.2)
    .map { (left, right) => ... }

result = joined2.groupBy(_.3)
    .reduceGroup(_.maxBy(_.2))
```

(3) Grouping/Aggregation reuses the partitioning from step (1) - no shuffle
What about Iteration?

- Iterate
- Delta Iterate
Iterative Algorithms

- **Algorithms that need iterations:**
  - Clustering, e.g., K-Means
  - Gradient descent, e.g., Logistic Regression
  - Graph algorithms, e.g., PageRank
  - ...

Loop over the working data multiple times.
Iteration

- **Loop** over the working data multiple times.

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

- In each *iteration*, the step function consumes the *entire input*, and computes the *next* version of the partial solution.

- A new version of the *entire* model in each iteration.
### Bulk Iteration - Example

<table>
<thead>
<tr>
<th>// 1st</th>
<th>2nd</th>
<th>10th</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(1) → 2</td>
<td>map(2) → 3</td>
<td>map(10) → 11</td>
</tr>
<tr>
<td>map(2) → 3</td>
<td>map(3) → 4</td>
<td>map(11) → 12</td>
</tr>
<tr>
<td>map(3) → 4</td>
<td>map(4) → 5</td>
<td>map(12) → 13</td>
</tr>
<tr>
<td>map(4) → 5</td>
<td>map(5) → 6</td>
<td>map(13) → 14</td>
</tr>
<tr>
<td>map(5) → 6</td>
<td>map(6) → 7</td>
<td>map(14) → 15</td>
</tr>
</tbody>
</table>
Bulk Iteration - Example

// 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

val input: DataSet[Int] = ...
def step(partial: DataSet[Int]) = partial.map(a => a + 1)
val numIter = 10;
val iter = input.iterate(numIter, step)
• Only parts of the model change in each iteration.
Delta Iteration - Example

Initial Input

1
2
3
4

1st Iteration

\[ \min(1, 2) \]
\[ \min(1, 2, 3, 4) \]

2
3
4

2nd Iteration

\[ \min(2, 3, 4) \]
\[ \min(2, 3, 4) \]

3
4

3rd Iteration

\[ \min(1, 2) \]
\[ \min(1, 2) \]

1
2

1

In Workset for next iteration

Not in Workset for next iteration

5
6
7

min(5, 6, 7)

min(5, 6, 7)

min(5, 6, 7)

min(5, 6, 7)

5
6
7

5
6
7

5
6
7

5
6
7
- **Computations** performed in each iteration for connected communities of a social graph.
Stratosphere
Execution Engine
**Stratosphere Architecture**

- **Master-worker model**

- **Job Manager**: handles job **submission** and **scheduling**.

- **Task Manager**: executes tasks received from JM.

- All operators start **in-memory** and gradually go **out-of-core**.
Jobs are expressed as data flows.

Job graphs are transformed into the execution graph.

Execution graphs consist information to schedule and execute a job.
Channels

- **Channels** transfer serialized records in buffers.

- **Pipelined**
  - Online transfer to receiver.

- **Materialized**
  - Sender writes result to disk, and afterwards it transferred to receiver.
  - Used in recovery.
Fault Tolerance

- Task failure compensated by backup task deployment.

- Track the execution graph back to the latest available result, and recomputes the failed tasks.
<table>
<thead>
<tr>
<th>Paradigm</th>
<th>MapReduce</th>
<th>Iterative Data Model</th>
<th>Distributed Collections (RDD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Model</td>
<td>Key/Value Pairs</td>
<td>Tuples</td>
<td>Key/Value Pairs</td>
</tr>
<tr>
<td>Runtime</td>
<td>Batch Processing</td>
<td>Streaming in-memory and out-of-core</td>
<td>Batch Processing in-memory and out-of-core</td>
</tr>
<tr>
<td>Compilation Optimization</td>
<td>None</td>
<td>Holistic Planning for Data Exchange, Sort/Hash, Caching, ...</td>
<td>?</td>
</tr>
</tbody>
</table>
http://stratosphere.eu