Big Data looks tiny from Stratosphere

Kostas Tzoumas
kostas.tzouumas@tu-berlin.de
What’s new

- Traditional databases can handle most of “Big Data” setups in terms of volume
- Hadoop encourages cheap data collection in both hardware and software and data reuse
- Also, accelerates data usage and creates a paradigm shift within the company
- New “Big Data” analytical applications create the demand for new data management software
Why new data management systems?

<table>
<thead>
<tr>
<th>Feature</th>
<th>NoSQL - Hadoop</th>
<th>NewSQL - Impala</th>
<th>Stratosphere</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Large aggregation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ETL</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SQL</td>
<td>✗ = Hive</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Complex analytics</td>
<td>✗ = Mahout</td>
<td>✗ = Madlib</td>
<td>✔</td>
</tr>
</tbody>
</table>

- Page Rank (iterated Matrix/Vector multiplication): \( p = A \cdot p \)
Stratosphere in one slide

Apache-licensed open source system

1. Declarative language

2. Embraces external data sources (HDFS)

3. Rich programming model (beyond MapReduce)

4. User-defined functions

5. Automatic parallelization and optimization

6. Iterative programs for complex analytics

7. Efficient and scalable execution engine
Outline

Stratosphere architecture

Program compilation

Program execution

Programming language and model
Sky program

val orderLines = filteredOrders join lines
  on { _.id } isEqualTo { _.orderId }
  map { (o, li) => OrderTotal(o.id, o.shipPr, li.price) }

val orderTotals = orderLines
  groupBy { pi => (pi.orderId, pi.shipPriority) }
  combine { _ reduce addRevenues }

Pact program

Sink 1

Reduce (on A)
  sum(B), avg(C)

Match (A = D)
  if (A > 3) emit

Map
  C := max(A, B)

Source 1
  Extract (A, B)

Source 2
  Extract (D, E)

Extended Scala compiler

Stratosphere optimizer

Picks data shipping and local strategies, operator order, generates runtime code

Job graph

Runtime

Hash- and sort-based out-of-core operator implementations, memory management

Execution graph

Nephele execution engine

Task scheduling, network data transfers, resource allocation, checkpointing
Sky program

```scala
val orderLines = filteredOrders join lines
  on { _.id } isEqualTo { _.orderId }
map { (o, li) => OrderTotal(o.id, o.shipPr, li.price) }

val orderTotals = orderLines
  groupBy { pi => (pi.orderId, pi.shipPriority) }
combine { _ reduce addRevenues }
```

Pact program

```
Sink 1

Reduce (on A) 
  sum(B), avg(C)

Match (A = D) 
if (A>3) emit

Map C := max(A,B)
if (D>4) emit

Source 1 
Extract (A,B)
Source 2 
Extract (D,E)
```

Runtime plan

- `Sink 1`
- `Match (A = D)`
  - if (A>3) emit
- `Reduce (on A)`
  - sum(B), avg(C)
- `buildHT (A)`
- `partition (A)`
- `Map C := max(A,B)`
  - if (D>4) emit
- `Source 1` 
  - Extract (A,B)
- `Source 2` 
  - Extract (D,E)

Job graph

- `F`
- `E`
- `D`
- `B`
- `C`
- `A`

Execution graph

- `Sink 1`
- `Reduce (on A)`
  - sum(B), avg(C)
- `Match (A = D)`
  - if (A>3) emit
- `Map C := max(A,B)`
- `if (D>4) emit`

Runtime

- Hash- and sort-based out-of-core operator implementations, memory management

Stratosphere optimizer

- Picks data shipping and local strategies, operator order

Nephele execution engine

- Task scheduling, network data transfers, resource allocation, checkpointing
What will be the SQL of NoSQL analytics?

- SQL will be the SQL of NoSQL analytics
- SQL and Jaql will be the SQL of NoSQL analytics
- Many languages depending on trends, requirements, background, and luck will be the SQL of NoSQL analytics
- SQL plus non-linear recursion and limited support for nested data will be the SQL of NoSQL analytics
Existing languages

- Are restricted to a flat or semi-structured data model (SQL, Hive, Pig, Jaql ...)
- Are restricted to a set of operators with given semantics (join, select, project, ...)
- More complex analysis is done by user-defined functions outside the language and is unknown to the compiler
- Can only express one-pass algorithms - no support for iterations
Data Scientist: The Sexiest Job of the 21st Century

A = load 'WordcountInput.txt';
B = MAPREDUCE wordcount.jar store A into 'inputDir' load 'outputDir' as
(word:chararray, count: int) `org.myorg.WordCount inputDir outputDir`;
C = sort B by count;

This is so 90s (or 00s)
Algorithm 1: Connected Components as a PACT Program

1 val vertices = ... // vertices: (vid, vid)
2 val edges = ... // define edges: (source, target)
3 // define the function that is iteratively evaluated
4 def incr = (s:Stream[Int, Int], ws:Stream[Int]) => {
5   // join the workset (changed vertices) with edges
6   val all = ws join edges on { . 1 } isEqualTo { . 1 }
7   using (w, e) ⇒ e._2 → w._2
8   // find the minimal candidate component id per vertex
9   val min = all reduceBy { . 1 }
10  using cs ⇒ cs minBy { . 2 }
11 // join candidates with solution set
12  val delta = min join on [_.1] isEqualTo [_.1]
13  using (n, s) ⇒ (n, s) match {
14    case ((v, cNew), (_, cOld))
15      if cNew < cOld ⇒ Some((v, cNew))
16    case _ ⇒ None
17  }
18  // return delta and new workset (identical here)
19  (delta, delta)
20 }  
21 // evaluate the function iteratively and assign the result
22 output ← incr iterate (s0=vertices keyBy {_.1}, ws0=vertices)
Sky: data analysis on Scala

- Feels like native Scala programming - data-parallel constructs (map, reduce, join, iterate, ...) and UDFs in same language, native type system
- Increased developer productivity by orders of magnitude less code
- Base for DSL development for analysts (Scala = Scalable Language)
- General yet optimizable - mapping to runtime type system with the help of code analysis
- Staged (extensible) compiler as implementation bases
val input = DataSource("<file>", RecordDataSourceFormat[Int,String]("\n", ","))
val filtered = input map {
  case (n, w) => if (n / 2 == 0) { (n, w + " check") } else { (-1, "false") }
}
val sink = filtered.write("<file>", RecordDataSinkFormat("\n", ","))

val input = DataSource(...)
val stub = new MapStub {
  private val serializer: ...
  val open(config: Configuration) {
    serializer = ... // init custom serializer
  }
  val map(in: ByteBuffer, out: ByteBuffer) {
    val result = if (in.readInt(0) / 2 == 0) {
      (in.readInt(0), in.readString(1) + " check")
    } else {
      (-1, "false")
    }
    serializer.write(out, result)
  }
}

val filtered = val MapContract(stub)
filtered.setInput(input)
val sink = val DataSink(...)
sink.setInput(filtered)
Meteor language

1 using ie;
2 using cleansing;
3
4 $articles = read from 'news.json';
5 $articles = annotate sentences $articles
6  use algorithm 'morphAdorner';
7 $articles = annotate entities $articles
8  use algorithm 'regex' and type 'person';
9 $peopleInNews = pivot $articles around
10  $person = $article.annotations[*].
11  entity
12  into {
13    name: $person,
14    articles: $articles
15  };
16
17 $persons = read from 'person.json';
18 $persons = remove duplicates
19  where average(levenshtein(name),
20    dateSim(birthDay)) > 0.95
21  retain longest(name);
22 $personsInNews = join $refPerson in
23  $persons,
24  $newsPerson in $peopleInNews
25  where $refPerson.name == $newsPerson.
26  name
27  into {
28    $refPerson.*,
29    articles: $newsPerson.articles[*].url
30  };
31
32 write $personsInNews to 'result.json';

- An extensible language for semi-structured data analysis
- JSON data, packages for information extraction, data cleansing, relational data processing
- Stratosphere as a compilation target for many languages
- Also working on: Pig (KTH), XQuery (Inria), SQL (TUB)
Sky program

val orderLines = filteredOrders join lines
  on { _.id } isEqualTo { _.orderId }
map { (o, li) => OrderTotal(o.id, o.shipPr, li.price) }

val orderTotals = orderLines
groupBy { pi => (pi.orderId, pi.shipPriority) }
combine { _ reduce addRevenues }

Pact program

val orderTotals = orderLines
groupBy { pi => (pi.orderId, pi.shipPriority) }
combine { _ reduce addRevenues }

Runtime plan

Stratosphere optimizer

Picks data shipping and local strategies, operator order

Job graph

Execution graph

Nephele execution engine
Task scheduling, network data transfers, resource allocation, checkpointing

Runtime

Hash- and sort-based out-of-core operator implementations, memory management
MapReduce

A directed acyclic graph consisting of exactly of two operators (map and reduce) on key-value pairs in a fixed order.
A **directed acyclic graph** consisting of exactly of two operators (map and reduce) on key-value pairs in a fixed order.
A directed acyclic graph consisting of exactly of two operators (map and reduce) on key-value pairs in a fixed order
An acyclic Pact program is an arbitrary dag of operators.

An operator consists of a second-order function and a first-order user-defined function on records.

Improved programmability and performance.

A directed acyclic graph consisting of exactly of two operators (map and reduce) on key-value pairs in a fixed order.
Need for iterations

- Relational Queries (uncorrelated)
- XQueries (uncorrelated)
- Single Pass Graph Algorithms
- Step Functions for Clustering
- Full Clustering Algorithms
- Multi-Pass Graph Algorithms
- Optimization
- Factorization of Matrices

Bulk iterations

$\text{while } S < f(S) \quad S = f(S)$

**constant data path:** executed only once (typically cached)

**dynamic data path:** executed many times

5.1 Link Analysis, Centrality and Community

- **Power Method**
- **Random Walk With Restart**
- **Katz centrality**
- **PageRank** models the web as a Markov chain, computes the relevance of web pages based on the underlying link structure.

5.2 Path Enumeration Problems in Graphs

- **Single-source maximum reliability**
- **Minimum spanning tree**
- **Single-source reachability**
Graphs are not so special

Specialized systems are re-developing core database concepts: operations on byte arrays, spilling to disk, fault tolerance, etc

**Instead:**

Avoid code repetition and data movement between systems by “teaching” iterations to a parallel data flow engine - with comparable performance
**But:** Bulk iterations do not exploit sparse computational dependencies inherent in graph and other algorithms.
Workset iterations

while \( W \neq \emptyset \)
\[
D = u(S, W)
\]
\[
W = \delta(D, S, W)
\]
\[
S = S \cup D
\]

Define \( W \): what might change

Decompose \( f \): produce delta \( D \) and next \( W \)

Internally:
Solution stored as partitioned (cached) index structure
Merge of \( S \) and \( D \) can be realized as index merge
## Comparison

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>MapReduce</th>
<th>Pregel</th>
<th>Stratosphere/Naiad</th>
<th>GraphLab</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fixed Functions – Map and Reduce</td>
<td>Supersteps over a data graph with messages passed</td>
<td>Iterative dataflow with operators and UDFs</td>
<td>Data graph with shared data table and update functions</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Concurrent execution of tasks within map and reduce phases</td>
<td>Concurrent execution of user functions over vertices within a superstep</td>
<td>Concurrent execution of operators during a stage</td>
<td>Concurrent execution of non-overlapping scopes, defined by consistency model</td>
</tr>
<tr>
<td>Data Handling</td>
<td>Distributed file system</td>
<td>Distributed file system</td>
<td>Flexible data channels: Memory, Files, DFS etc.</td>
<td>Undefined – Graphs can be in memory or on disk</td>
</tr>
<tr>
<td>Task Scheduling</td>
<td>Fixed Phases – HDFS Locality based map task assignment</td>
<td>Partitioned Graph and Inputs assigned by assignment functions</td>
<td>Job and Stage Managers assign operators to available daemons/tasks</td>
<td>Pluggable schedulers to schedule update functions</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>DFS replication + Task reassignment / Speculative execution of Tasks</td>
<td>Checkpointing and superstep re-execution</td>
<td>Operators/Task failure recovery</td>
<td>Synchronous and asynchronous snapshots</td>
</tr>
<tr>
<td>Developed by</td>
<td>Google</td>
<td>Google</td>
<td>TU Berlin / Microsoft</td>
<td>Carnegie Mellon</td>
</tr>
</tbody>
</table>

“Scalable Machine Learning for Big Data” tutorial at ICDE 2012
Sky program

```scala
val orderLines = filteredOrders join lines
  on { _.id } isEqualTo { _.orderId }
map { (o, li) => OrderTotal(o.id, o.shipPr, li.price) }

val orderTotals = orderLines
  groupBy { pi => (pi.orderId, pi.shipPriority) }
  combine { _ reduce addRevenues }
```

Pact program

```scala
Sink 1

Reduce (on A)
  sum(B), avg(C)

Map
  C := max(A,B)

Source 1
  Extract (A,B)

Map
  if (D>4) emit

Source 2
  Extract (D,E)
```

Runtime plan

```
Sink 1

Match (A = D)
  if (A>3) emit

Reduce (on A)
  sum(B), avg(C)

Map
  C := max(A,B)

Source 1
  Extract (A,B)

Map
  if (D>4) emit

Source 2
  Extract (D,E)
```

Stratosphere optimizer

*Picks data shipping and local strategies, operator order*

Runtime

*Hash- and sort-based out-of-core operator implementations, memory management*

Nephele execution engine

*Task scheduling, network data transfers, resource allocation, checkpointing*
Why optimize?

Order?

What type of join?
Optimization in Stratosphere

Optimize plans with parallel UDFs à la relational databases (using static code analysis)

Cost-based decision of parallel execution
Parallel databases 101

sort-merge or hash joins and reducers

broadcast or partition joins

late or early materialization (cost-based)
Mapping of Scala data types to (native) operations on raw byte arrays

Compiler techniques to optimize iterative data analysis programs

Operator Strength Reduction
KEITH D. COOPER
Rice University
L. TAYLOR SIMPSON
BOPS, Incorporated
and
CHRISTOPHER A. VICK
Sun Microsystems, Incorporated

Static code analysis of user code to discover properties for database-style optimization
Sky program

```scala
val orderLines = filteredOrders join lines
  on { _.id } isEqualTo { _.orderId }
  map { (o, li) =>
    OrderTotal(o.id, o.shipPr, li.price)
  }

val orderTotals = orderLines
  groupBy { pi => (pi.orderId, pi.shipPriority) }
  combine { _ reduce addRevenues }
```

Pact program

```
Sink 1

Reduce (on A)
  sum(B), avg(C)

Match (A = D)
  if (A > 3) emit

Map
  C := max(A, B)

Source 1
  Extract (A, B)

Source 2
  Extract (D, E)
```

Extended Scala compiler

Stratosphere optimizer

*Picks data shipping and local strategies, operator order*

Nephele execution engine

*Task scheduling, network data transfers, resource allocation, checkpointing*
Stratosphere runtime

- An efficient runtime for UDF-centric programs
- Written in Java to avoid extensive switches between runtime and UDF code
- Implements its own memory manager to avoid Java object overhead and garbage collections
- Native operators work directly on serialized data
- Based on a data streaming (pipelining) engine
Nephele engine

- Schedules parallel tasks that contain arbitrary computations, and channels that carry data between tasks
- Permits both “push” model for pipelined operators and “pull” model
The reasons for that are twofold. First, Stratosphere's push-merger that compiles and executes SQL-like queries as se-
clauses are removed and the filter parameters for the mar-
sequences of Hadoop MapReduce jobs. In the modified ver-
time in the Word Count map phase goes into tokenizing the
improvement.
We also note that a substantial amount of the processing
fashion [34]. This optimization is especially useful for keys
The same optimization can be done for the Hadoop
input
The same optimization can be done for the Hadoop

(c) TPCH Q3

(d) Triangle Enumeration

(f) Connected Components
Trends and next steps
From “Big Data” to “Fast Data” - large scale distributed data streaming with iterative UDFs

What will be the data analytics language of the future

How to exploit and influence the design of new hardware for Big Data Analytics

Real world use cases that demand a replacement Hadoop MapReduce
We are building Stratosphere, the next-generation platform for rich analytics on big data

www.stratosphere.eu
Fork it at github.com/dimalabs/ozone