Performance Optimization Techniques and Tools for Data-Intensive Computation Platforms

Licentiate Seminar Presentation
June 11, KTH Kista, Sweden

Vasiliki Kalavri
kalavri@kth.se
Outline

1. Introduction & Research Objectives
2. Background
3. Contributions
   3.1. State-of-the-Art Survey on MapReduce
   3.2. Integration of Pig and Stratosphere
   3.3. Approximate Results in MapReduce
   3.4. Results Materialization and Reuse
   3.5. Asymmetry in Large-Scale Graph Analysis
4. Conclusions & Future Work
Introduction
The “Big Data” Era

"information that can't be processed or analyzed using traditional processes or tools"

Understanding Big Data, Book by IBM Research, 2012

"data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time"

The Pathologies of Big Data, Adam Jacobs, Communications of the ACM
Big Data Use-Cases
The 3 Vs of Big Data

- **Volume**: Facebook reports 15TB of raw compressed data per day (in 2010!)
- **Variety**: Relational data, xml, json, clicks, social interactions, purchase logs, raw text etc.
- **Velocity**: 100 hours of video are uploaded to YouTube every minute; 58 million tweets per day
Big Data processing is *slow*

- MapReduce-like batch processing can be 10x or more slower than RDBMS
- Jobs may take several hours before they return a result

original comic at: [http://xkcd.com/303/](http://xkcd.com/303/)
Research Objective

To make data-intensive computation platforms faster, by reducing:

- the dataset sizes
- the amount of computation

Avoid redundancy in data and computation
Research Questions

Can we improve performance while:

● maintaining application transparency?
● being system-independent?
● keeping user involvement to the minimum?

What are the trade-offs?
Background
Common design choices

- *Shared-nothing* clusters of commodity machines
- Programming models that express *data-parallelism* and abstract communication
- *Fault-tolerant* distributed storage
- Incentive to maximize *data locality*
Typical Big Data Architecture

**Programming Model**
- high-level APIs
- ML DSL
- graph processing libraries
  - operators
  - data types
  - I/O utilities

**Runtime**
- compiler
- planner/optimizer
- scheduler
- resource manager
- execution engine

**Persistent Storage**
- file system
- database
- key-value store
MapReduce/Hadoop

● A programming abstraction and open-source implementation that transparently handles
  ○ parallelization
  ○ data distribution
  ○ fault-tolerance

● Inspired by the functional map and reduce primitives

● Operates on key-value pairs
MR Programming Model
Apache Pig

- A high-level dataflow system
  - A high-level language: Pig Latin
  - A set of compilers that generate MapReduce jobs
Why Pig?

• **Simple language: Pig Latin**
  ○ easy for users familiar with SQL/scripting
  ○ lower cost to write and maintain

• **Common Operators already provided**
  ○ JOIN, GROUP, FILTER, SORT

• **Scalability and Fault-tolerance of Hadoop**
Plan Compilation

Logical Plan

Physical Plan

MapReduce Plan
In original MR, a reducer task cannot fetch the output of a map task which hasn't committed its output to disk.
MROnline Advantages

- **Better performance through pipelining**
  - avoid frequent materialization
  - improve reducers’ response times
- **Supports online aggregation**
  - return partial results
- **Improves utilization**
  - reduce tasks do not block waiting for mappers to finish (in the same job)
Stratosphere

- A distributed data processing system
- Directed Acyclic Graphs (DAGs) of sources, sinks and operators
- Abstracts data partitioning, parallelism, communication
The Stratosphere Stack
Contributions
Thesis Contributions

1. State-of-the-Art Survey on MapReduce
2. Integration of Pig and Stratosphere
3. Approximate Results in MapReduce
4. Results Materialization and Reuse
5. Asymmetry in Large-Scale Graph Analysis
MapReduce: Limitations, Optimizations and Open Issues

Vasiliki Kalavri, Vladimir Vlassov
17 July 2013, Melbourne, Australia
Motivation

● Numerous Hadoop variations and enhancements over the past few years
  ○ each branching out from vanilla Hadoop
  ○ hard to choose the appropriate tool
  ○ no categorization / classification exists

● In our survey
  ○ overview existing variations
  ○ classify the optimizations
  ○ identify trends and open issues
MapReduce Limitations

- **Performance**
  - initialization, scheduling, coordination
  - data materialization - intensive disk I/O

- **Programming Model**
  - single-input operators
  - fixed processing pipeline - job chaining
  - no support for iterations

- **Configuration and Automation**
  - sensitive to configuration parameters
  - complicated tuning
## Systems Summary

<table>
<thead>
<tr>
<th>System</th>
<th>Major Contribution</th>
<th>Open-Source, Available?</th>
<th>Transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR Online</td>
<td>Pipelining, Online aggregation</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>EARL</td>
<td>Fast approximate results</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hadoop++, HAIL</td>
<td>Improve relational operations</td>
<td>no</td>
<td>yes / no</td>
</tr>
<tr>
<td>MRShare</td>
<td>Concurrent work sharing</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ReStore</td>
<td>Reuse of previous computations</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SkewTune</td>
<td>Automatic skew mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>CoHadoop</td>
<td>Data colocation</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>HaLoop</td>
<td>Iterations support</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Incoop</td>
<td>Incremental processing</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Starfish</td>
<td>Dynamic self-tuning</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Sailfish</td>
<td>I/O minimization, automatic tuning</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Manimal</td>
<td>Automatic data-aware optimizations</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Open Issues

● No standard benchmark
● No "typical" MapReduce workload
● Each system is evaluated using different
  ○ datasets
  ○ applications
  ○ deployments
    ■ impossible to compare or only compare with vanilla Hadoop

● Application transparency
PonIC: Using Stratosphere to Speed Up Pig Analytics

Vasiliki Kalavri\textsuperscript{1}, Vladimir Vlassov\textsuperscript{1}, Per Brand\textsuperscript{2}

\textsuperscript{1}KTH, \textsuperscript{2}SICS

30 August 2013, Aachen, Germany
Motivation

The limitations and inflexibility of Hadoop are still present in the backend...

- **Replace the execution engine!**

Diagram:
- **single input operators**
- **materialization**
- **fixed pipeline**
PonIC: Pig on Input Contracts

- Pig Latin on Stratosphere
  - translates Pig Logical Plans to Stratosphere DAGs
  - benefits from Stratosphere’s flexibility and efficiency
  - benefits from the optimizer and the automatic physical selection strategies

application transparent
Plan Optimization in PonIC

- Simpler translation process
- Simpler final plans
- No need for load-store “glue” between jobs
- 2 levels of plan optimization
PonIC vs. Pig/MapReduce
Conclusions

- Hadoop’s 2-stage fixed pipeline and frequent materialization harm Pig’s performance
- Pig can run on another execution engine
  - without loss of expressiveness
  - with possible performance gains
- Integration with a different backend is possible after the Logical Plan layer
Problem and Motivation

Big data processing is usually very time-consuming...

... but many applications require results really fast or can only use results for a limited window of time

Luckily, in many cases results can be useful even before job completion

- tolerate some inaccuracy
- benefit from faster answers
Online Aggregation

- Apply the reduce function to the data seen so far
- Use % of input processed to estimate accuracy
Sampling Challenges

- **Data in HDFS**
  - Disk access is terribly slow
  - Random disk access for sampling is even slower

- **Unstructured Data**
  - Sample based on what?
  - We don’t know the query, we don’t know the key or the value!
The Block Sampling Technique

- Sample on the block-level, before the data reach the mapper tasks
- In-memory shuffling for better performance
Performance - Overhead

- Snapshot freq = 10%

- Graphs showing execution time vs. input size (GB) and effective sampling rate.
MapReduce Online vs. Block Sampling

Average Temperature Estimation on Weather Data

Each dot corresponds to one year (10 in total)

MapReduce Online
- has 100% error (no estimation at all) for some years
- provides an estimate for all 10 years, after processing 40% of the data
Conclusions

- Useful results even before job completion
- Disk random access is prohibitively expensive → efficiently emulate sampling using in-memory shuffling
- Higher sampling rate improves accuracy but also increases communication costs
Motivation

- Avoid *computational redundancies*
  - filter out bad records, spam e-mail
  - data representation transformations
- Microsoft has found a 30%-60% similarity in queries submitted for execution
- A Berkeley MapReduce workload characterization study shows a big need for *caching* job results
m2r2: materialize - match - rewrite - reuse

- A *language-independent* framework for
  - storing
  - managing and
  - using
  - previous job and sub-job results
- Operates on the *logical plan* level, in order to support different languages and backend execution engines
m2r2 Implementation

- Prototype on top of Pig/Hadoop
- Results Cache: HDFS
- Repository: MySQL Cluster
  - in-memory, highly-available and fault-tolerant
- Garbage Collection: separate module
  - policy on reuse frequency and last access time
Performance

Materialization Overhead
Execution time without reuse

Speedup re-using sub-jobs
Conclusions

- The logical plan is the proper layer to build a language-independent reuse framework.
- When there exists reuse opportunity, query execution time can be immensely reduced:
  - 65% on average in our experiments.
- The materialization overhead is rather small and I/O dominant.
Asymmetry in Large-Scale Graph Analysis, Explained

Vasiliki Kalavri\textsuperscript{1}, Stephan Ewen\textsuperscript{2}, Kostas Tzoumas\textsuperscript{2}, Vladimir Vlassov\textsuperscript{1}, Volker Markl\textsuperscript{2}, Seif Haridi\textsuperscript{1}

\textsuperscript{1}KTH, \textsuperscript{2}TU Berlin
Motivation

- Many of large-scale data processing applications include \textit{fixed point} iterations
  - social network analysis
  - web graph analysis
  - machine learning
Asymmetrical Convergence

- Often, in fixed point iterations, some elements converge faster than others.
- Not all elements require an update in every iteration.
Can we detect the elements that require recomputation and avoid redundant computations?
Contributions

- A categorization of optimizations for fixed point iterative graph processing
- Necessary conditions under which, it is safe to apply optimizations
- A mapping of existing techniques to graph processing abstractions
- An implementation of template execution plans

Optimized algorithms yield *order of magnitude* gains!
Asymmetry in Connected Components
Iterative Plans - Bulk

In each iteration, all elements are computed.
Iterative Plans - Dependency

In each iteration, only elements whose - at least one - neighbor has changed, are recomputed:

- in each iteration, only changed elements are emitted in W
- before executing the update function, we need to gather the rest of the dependencies (neighbors values) of the candidates
Iterative Plans - Incremental

In each iteration, only changed elements are recomputed, using only the changed neighbors.
Iterative Plans - Delta

In each iteration, only changed elements are recomputed, using only the *deltas* of changed neighbors.
## Iteration Techniques Equivalence

<table>
<thead>
<tr>
<th>Iteration Technique</th>
<th>Equivalent to Bulk?</th>
<th>Vertex Activation</th>
<th>Vertex Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bulk</td>
<td>n/a</td>
<td>always</td>
<td>using values of all in-neighbors</td>
</tr>
<tr>
<td>Dependency</td>
<td>always</td>
<td>if any in-neighbor is updated</td>
<td>using values of all in-neighbors</td>
</tr>
<tr>
<td>Incremental</td>
<td>step function idempotent and weakly monotonic</td>
<td>if any in-neighbor is updated</td>
<td>using values of changed in-neighbors</td>
</tr>
<tr>
<td>Delta</td>
<td>step function linear over composition operator</td>
<td>if any in-neighbor is updated</td>
<td>using values of changed in-neighbors</td>
</tr>
</tbody>
</table>
## Iteration Techniques Support in Graph Processing Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Bulk</th>
<th>Dependency</th>
<th>Incremental</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>GraphLab</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>GraphX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Powergraph</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Stratosphere</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- **X**: provided by default
- **X**: can be easily implemented
- **X**: possible, but non-intuitive
Performance - Connected Components

Execution Time

Connected Components - Livejournal

- Bulk
- Dependency
- Incremental

log(time)

# iteration
Performance - PageRank

Execution Time
PageRank - Livejournal

log(time)

Bulk
Dependency
Delta

# iteration

1 5 10 15 18
Conclusions

- Graph-processing algorithms can be efficiently implemented using common relational operators
- Asymmetrical convergence can be detected and if exploited, can lead to order of magnitude performance gains
- A cost model is needed to choose the most efficient execution plan
Conclusions & Future Work
Conclusions

In a data-intensive computation framework

- the amount of *data* and *computation* can be reduced by
  - using more efficient backends
  - sampling
  - caching and reuse
  - detecting “inactive” parts of the computation flow
Can we improve performance while:

- maintaining **application transparency**?
  - yes, when re-using the upper layers of the frameworks
  - challenging because most systems are tightly coupled with their backend engines
Can we improve performance while:

- **being system-independent?**
  - most of the time **yes**, as systems are very frequently designed based on similar primitives
  - challenging to find the correct level of abstraction
Can we improve performance while:

● keeping user involvement to the minimum?
  ○ yes, especially when working with high-level systems
  ○ challenging to automate configuration parameters depending on specific workloads
What are the trade-offs?

- when using an alternative backend
  - translation and compilation overheads, data transformations
- when sampling
  - results accuracy vs. response time
- when caching
  - materialization, storage and garbage collection overheads
  - matching and query re-writing
- when detecting inactive parts
  - tracking dependencies
Future Work

- Extend m2r2 to other high-level systems, such as Hive and Shark
- Design and implement a cost model for fixed point iterative algorithms
  - choose the most efficient plan at runtime
- Explore optimization opportunities in other frameworks and application domains
  - asynchronous execution, real-time graph processing
Acknowledgement

- Erasmus Mundus Joint Doctorate in Distributed Computing (emjd-dc) [http://emjd-dc.eu/](http://emjd-dc.eu/)

- My advisor, Vladimir Vlassov
- My secondary advisors, Christian Schulte and Peter Van Roy (UCL)
- Anne Hakansson
- My co-authors
- My colleagues at KTH and UCL
- Anonymous reviewers
Performance Optimization Techniques and Tools for Data-Intensive Computation Platforms

Licentiate Seminar Presentation
June 11, KTH Kista, Sweden

Vasiliki Kalavri
kalavri@kth.se