Data-Intensive Applications on HPC Using Hadoop, Spark and RADICAL-Cybertools

Shantenu Jha, Andre Luckow, Ioannis Paraskevakos
RADICAL, Rutgers, http://radical.rutgers.edu

Agenda

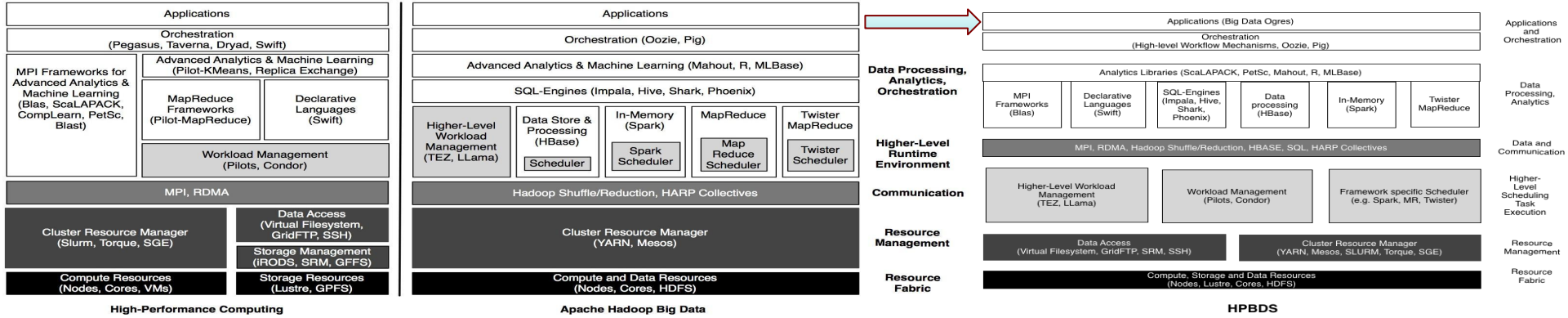
1. Motivation and Background
2. Pilot-Abstraction for Data-Analytics Application on HPC and Hadoop
3. Tutorial
4. Performance: Understanding Runtime Trade-Offs
5. Conclusion and Future Work

1.1 The Convergence of HPC and "Data Intensive" Computing

At multiple levels: Applications, Micro-Architectural ("near data computing" processors), Macro-Architectural (e.g. File Systems), Software Environment (e.g., Analytical Libraries).

Objective: Bring ABDS Capabilities to HPDC

- HPC: Simple Functionality, Complex Stack, High Performance ABDS: Advanced Functionality



A Tale of Two Data-Intensive Paradigms: Data Intensive Applications, Abstractions and Architectures In collaboration with Geoffrey Fox (Indiana), http://arxiv.org/abs/1403.1528

1.2 MIDAS: Middleware for Data-intensive Analysis and Science

- Application is integrated deeply with Infrastructure.
  - Great for performance. But bad for extensibility & flexibility.
- Multiple levels of functionality, indirection and abstractions.
  - Performance is often difficult.
- Challenge: How to find “Sweet Spot”?
  - “Neck of hour glass” for multiple applications and infrastructure.

1.2 MIDAS: Middleware for Data-intensive Analysis and Science

- MIDAS is the middleware for support analytical libraries, by providing
  - Resource management.
    - Pilot-Hadoop for managing ABDS frameworks on HPC
  - Coordination and communication.
    - Pilot In-Memory for supporting iterative analytical algorithms
  - Address heterogeneity at the infrastructure level
    - File and storage abstractions.
  - Flexible and multi-level compute-data coupling.
- Must have a well-defined API and semantics that can then be used by application and SPIDAL library/layer.

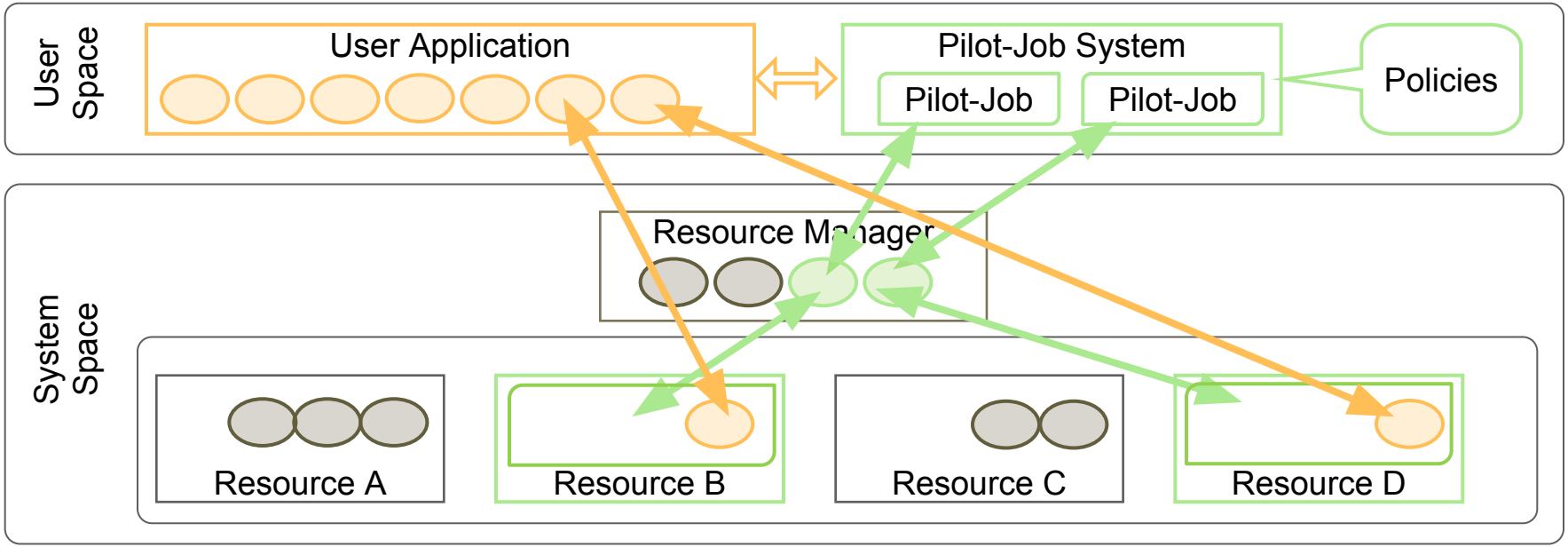
1.3 Application Integration with MIDAS

- Type 1: Some applications will require libraries before they need performance/scalability
  - Advantages of functionality and commonality
- Type 2: Some applications are already developed but need performance/scalability, i.e. have necessary functionality, but stymied by lack of scalability
  - Integration into MIDAS directly for performance
- Type 3: Once applications libraries have been developed, make high-performance by integrating libraries to underlying capabilities

Part II: Pilot-based Runtime for Data Analytics

2.1 Introduction Pilot Abstraction

Working definition: A system that generalizes a placeholder job to provide multi-level scheduling to allow application-level control over the system scheduler via a scheduling overlay.



2.1 Motivation Pilot-Abstraction

The Pilot-Abstraction provides a well-define resource management layer for MIDAS:

- Application-level scheduling well suited for fine-grained data parallelism of data-intensive applications
- Data-intensive applications more heterogeneous and thus, more demanding with respect to their resource management needs
- Application-level scheduling enables the implementation of a data-aware resource manager for analytics applications
- Interoperability Layer between Hadoop (Apache Big Data Stack (ABDS) and HPC

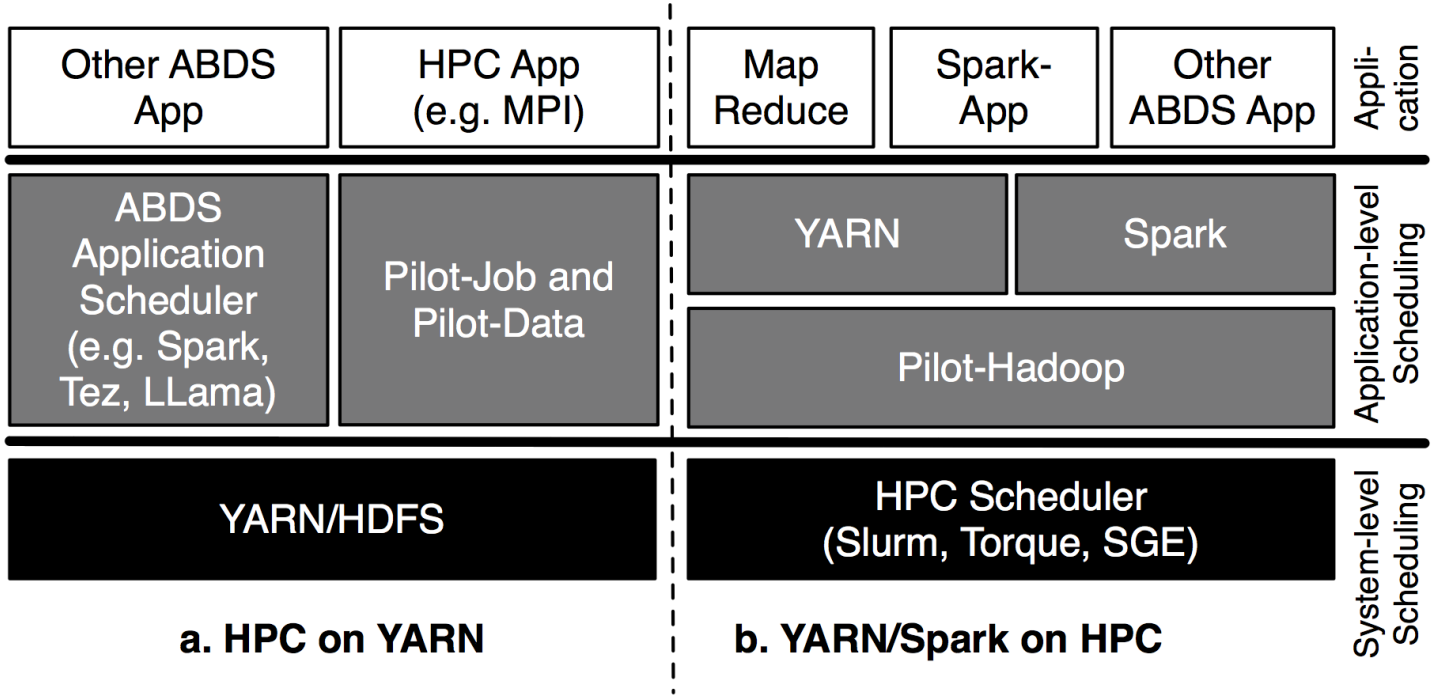
2.1 Motivation: Hadoop and Spark

Source: http://hadoop.apache.org

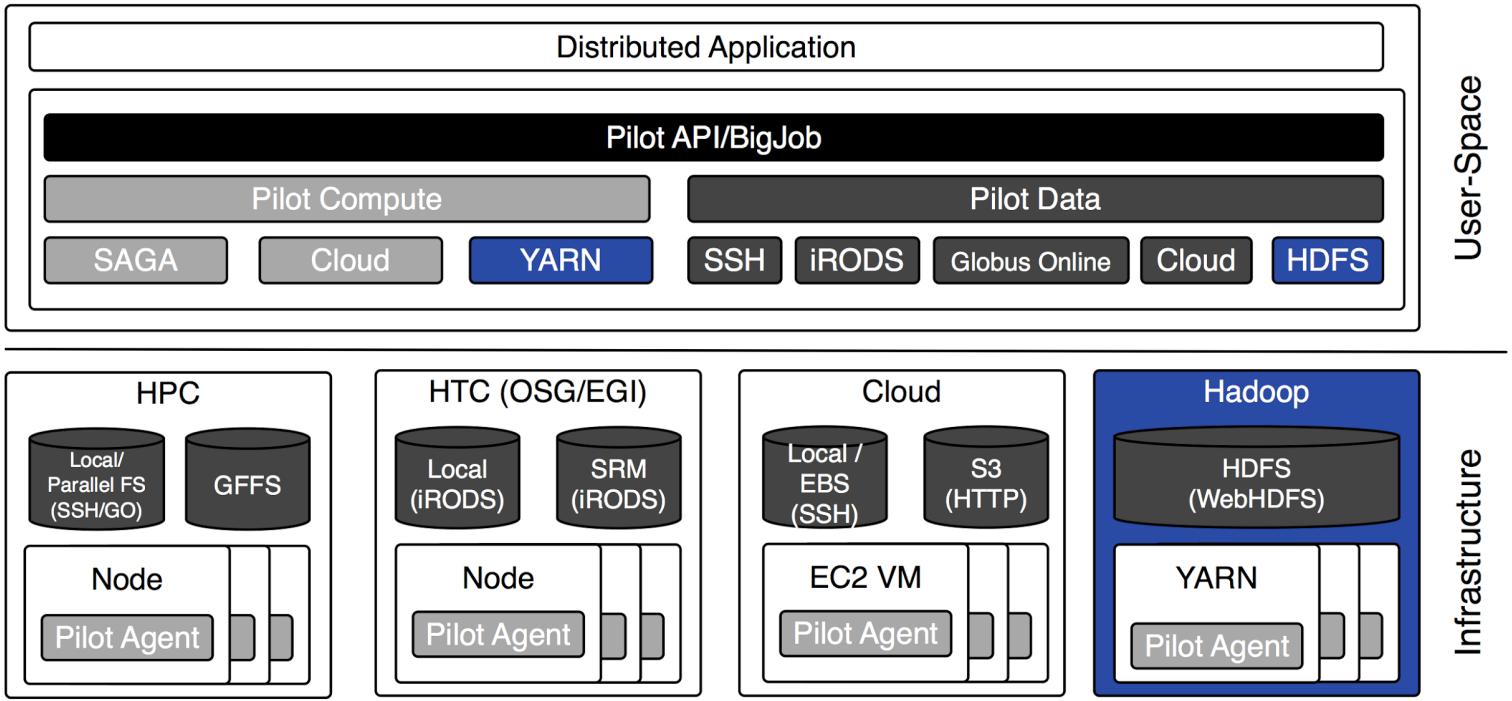
Source: http://spark.apache.org

- De-facto standard for industry analytics
- Manifold ecosystem with many different analytics tools, e.g. Spark MLLib, H20 (referred to as **Apache Big Data Stack (ABDS)**)
- Novel, high-level abstractions: SQL, DataFrames, Data Pipelines, Machine Learning

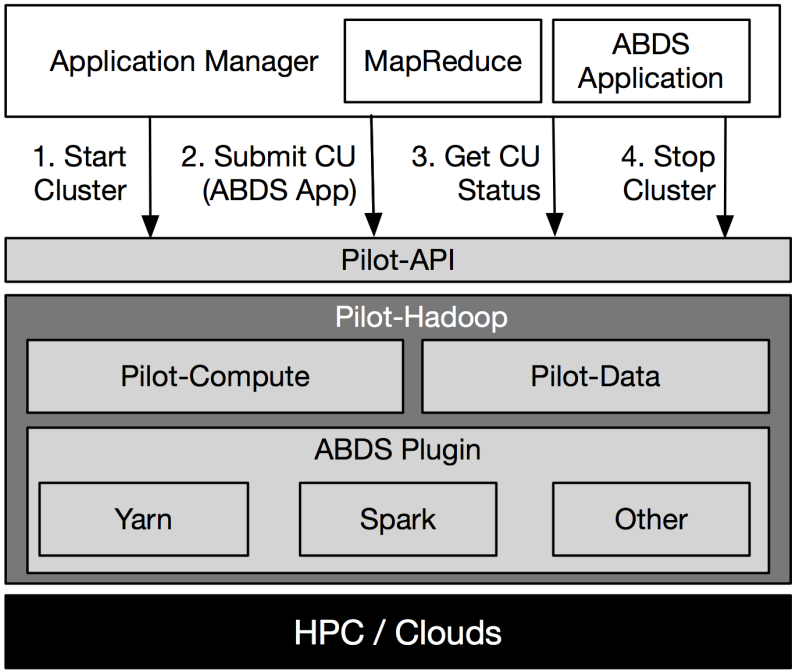
2.1 HPC and ABDS Interoperability



2.2 Pilot-Abstraction on Hadoop



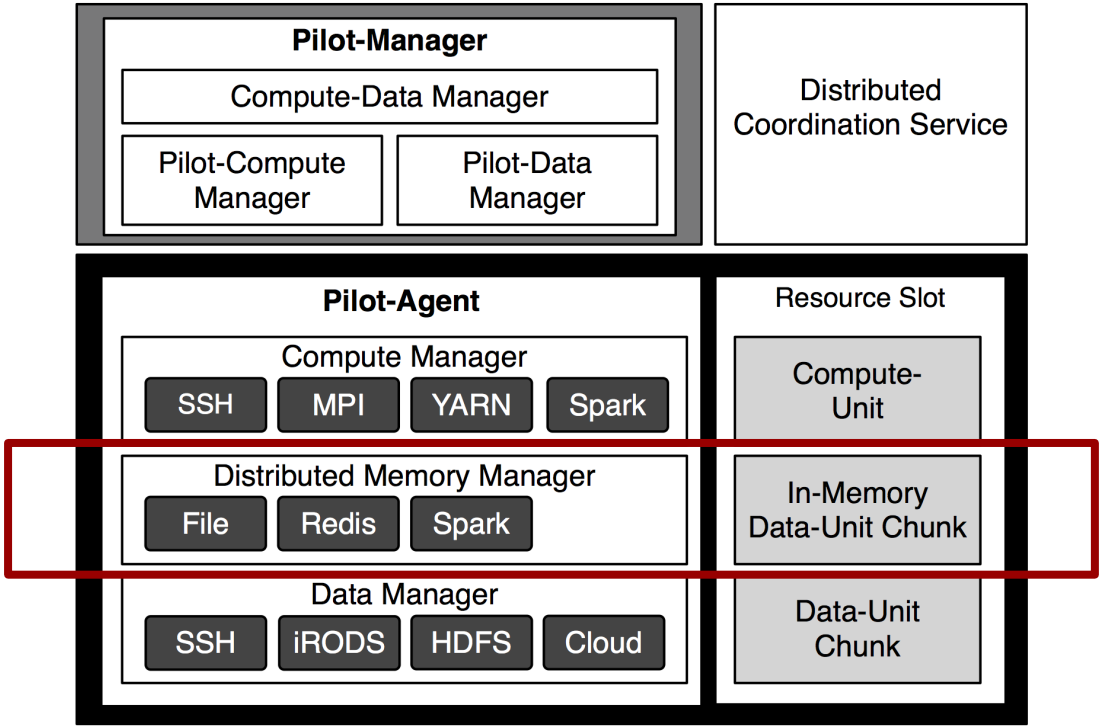
2.3 Pilot-Hadoop: ABDS on HPC



Pilot-Job is used for managing Hadoop Cluster

Pilot-Agent responsible for managing Hadoop resources: CPU cores, nodes and memory

2.4 Pilot-Memory for Iterative Processing.



Provide common API for distributed cluster memory

2.5 Abstraction in Action

2. Pilot-Spark and Pilot-InMemory Implementation

Setup Spark cluster on local machine or HPC resource

2.1 Start Spark Cluster using Pilot-Spark

see https://github.com/drelu/saga-hadoop

In [1]:
```
pilot_compute_description = {
                            "resource_url":"fork://localhost",
                            "number_cores": 1,
                            "cores_per_node":1,
                            "type":"spark"
                            }
pilot = PilotComputeService.create_pilot(pilot_compute_description);

# print out details of Pilot-Spark
details = pilot.get_details()
```

...

2.2. Utilize Spark Native

In [2]:
```
sc = pilot.get_spark_context()
sc.version

rdd = sc.textFile("/path/to/leaflet_graph")
```

Out[2]:
```
u'1.3.0'
```

2.3 Pilot-InMemory Implementation

In [1]:
```
du = DistributedInMemoryDataUnit(name="LeafletFinderGraph", sc=sc)
```

1. Run Spark or Hadoop on a local machine, HPC or cloud resource

2. Seamless access to native Spark features and libraries

3. Use Pilot-Data API

Part III: Tutorial

3. Tutorial

1. Pilot-Abstraction Introduction
2. Pilot-Hadoop
3. Advanced Analytics on HPC and BigData:
    a. KMeans
    b. Graph Analytics

see Github/iPython Notebook

Part IV: Performance: Understanding Runtime Trade-Offs

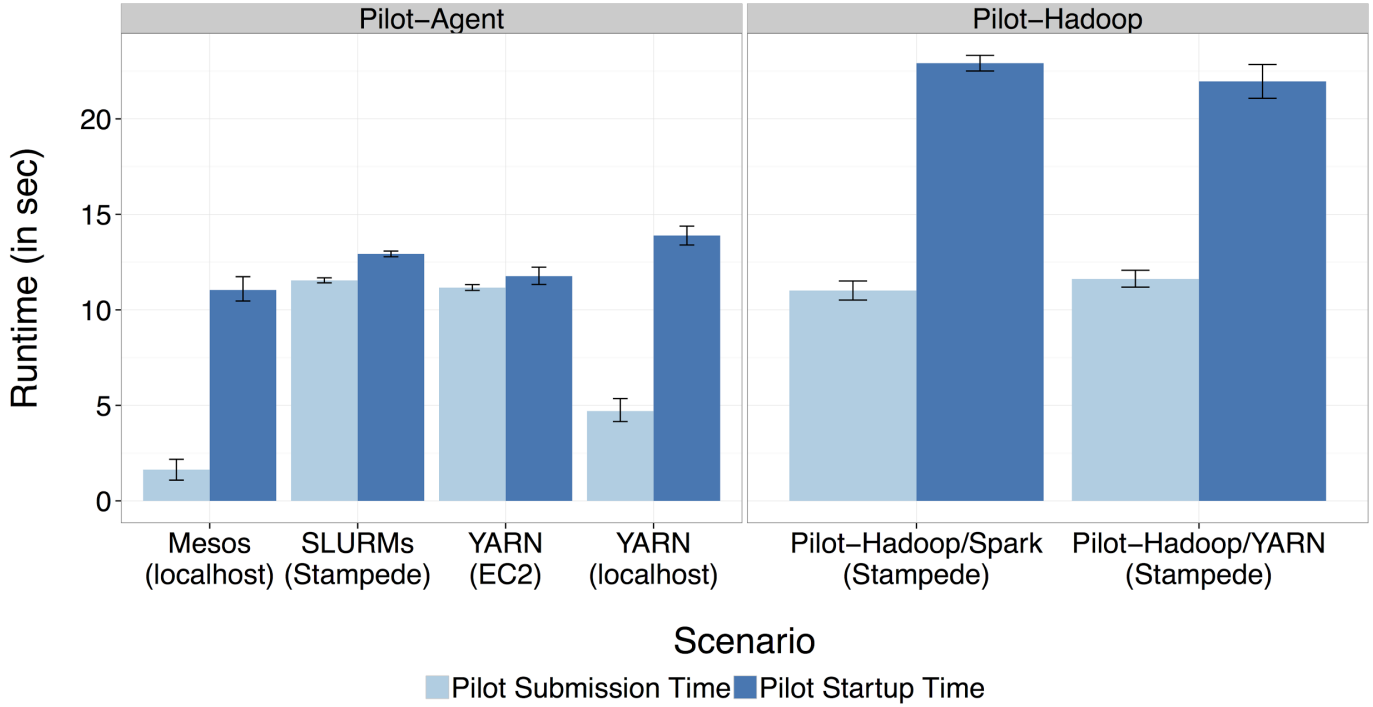
4. Performance

4.1 Overhead of Pilot-Abstraction

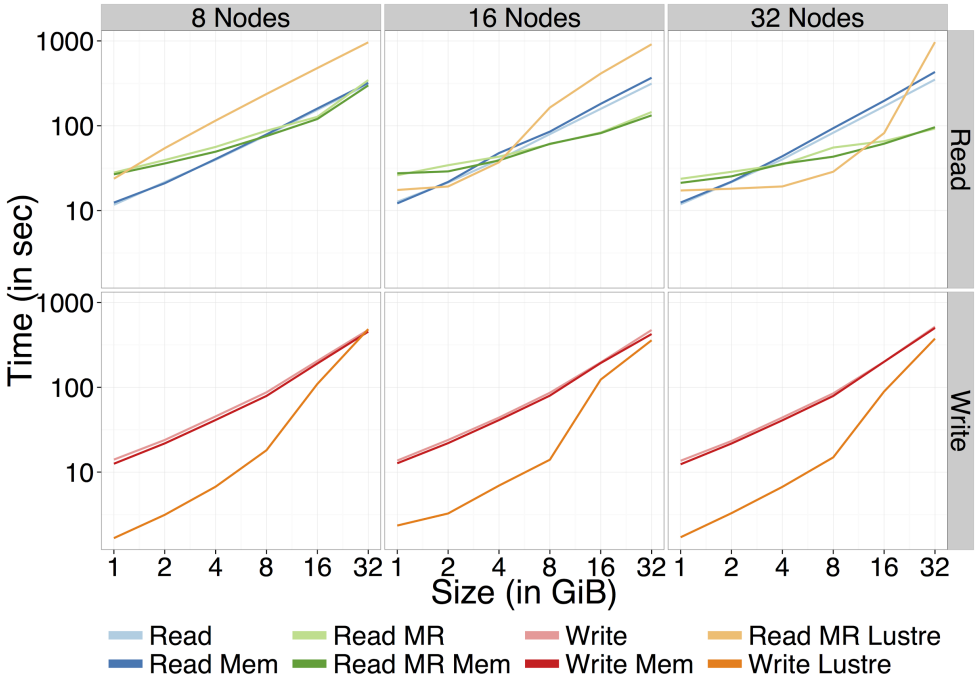
4.2 HPC vs. ABDS Filesystem

4.3 KMeans

4.1 Pilot-Abstraction Overhead



4.2 HPC vs. ABDS Filesystem



Lustre vs. HDFS on up to 32 nodes on **Stampede**

Lustre good for **medium-sized data**

Writes on Lustre faster - gap decreases with data size

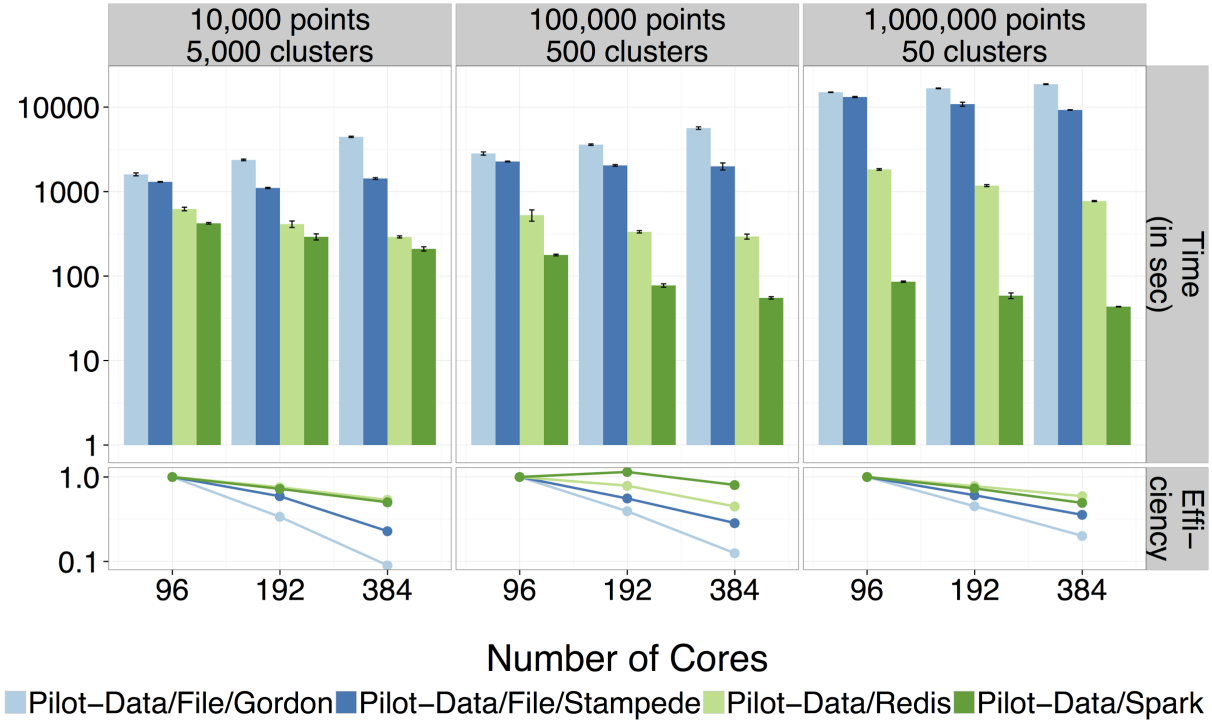
Parallel reads faster with HDFS

HDFS Memory option provides slight advantage

4.3 Pilot-Data on Different Backends

Managing heterogeneous HDFS Backends with Pilot-Data on different XSEDE resources

4.4 KMeans on Pilot-Memory



Part V: Conclusion, Future Work and Q&A

5. Conclusion and Future Work

Big Data application very heterogeneous

Complex infrastructure landscape with many layers of scheduling requires higher-level abstractions for reasoning.

Next Steps:

- Applications: Graph Analytics (Leaflet Finder)
- Application Profiling and Scheduling

Work-in-Progress Paper: http://arxiv.org/abs/1501.05041

5. Conclusions and Future Work

- Balanced the workload of each task in order to increase the task level parallelism
- Able to provide linear speedup
- Next Steps:
  - Ongoing experimentation to find the dependency on n_1.
  - Compare with ABDS method? If so, which?

Thank you

Data-Intensive Applications on HPC Using Hadoop, Spark and RADICAL-Cybertools

Shantenu Jha and Andre Luckow

The tutorial material is available as iPython notebook at:

- http://nbviewer.ipython.org/github/radical-cybertools/supercomputing2015-tutorial/blob/master/Tutorial%20Overview.ipynb (http://nbviewer.ipython.org/github/radical-cybertools/supercomputing2015-tutorial/blob/master/Tutorial%20Overview.ipynb)

The code is published on Github:

- https://github.com/radical-cybertools/supercomputing2015-tutorial (https://github.com/radical-cybertools/supercomputing2015-tutorial)

Requirements and Setup:

Python with the following libraries:

- Numpy
- Pandas
- Scikit-Learn
- Seaborn
- BigJob2

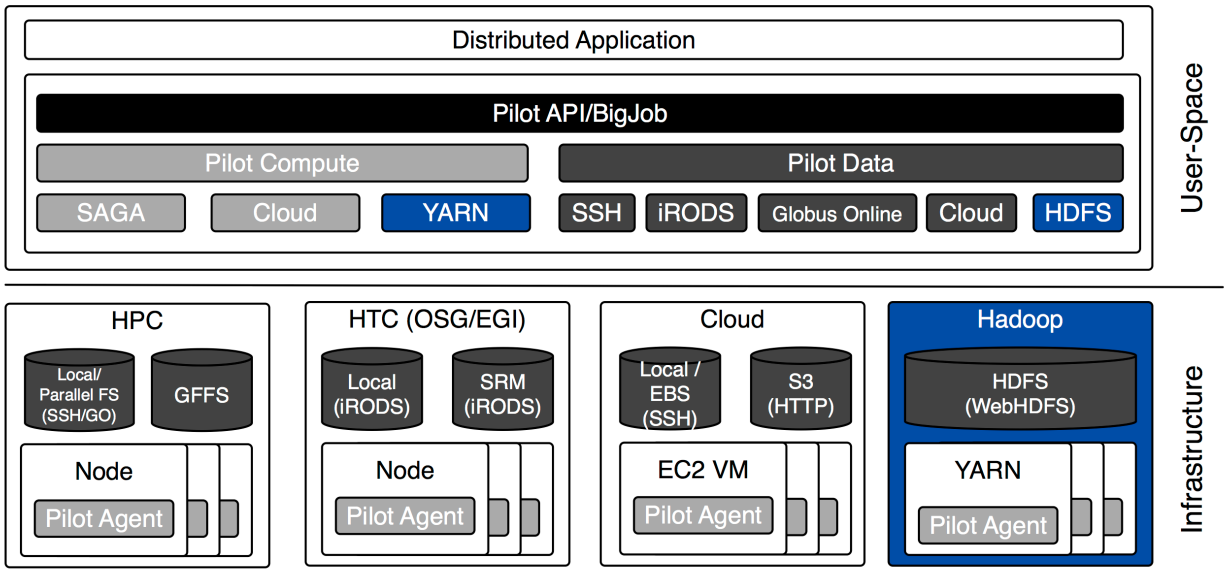
We recommend to use Anaconda (http://continuum.io/downloads).

1. Pilot-Abstraction for distributed HPC and Apache Hadoop Big Data Stack (ABDS)

The Pilot-Abstraction has been successfully used in HPC for supporting a diverse set of task-based workloads on distributed resources. A Pilot-Job is a placeholder job that is submitting to the resource management system and is used as a container for a dynamically determined set of compute tasks. The Pilot-Data abstraction extends the Pilot-Abstraction for supporting the management of data in conjunction with compute tasks.

1.1 Pilot-Abstraction

The Pilot-Abstraction supports a heterogeneous resources, in particular different kinds of cloud, HPC and Hadoop resources.



1.2 Example

The following example demonstrates how the Pilot-Abstraction is used to manage a set of compute tasks.

In [5]:
```
%matplotlib inline
import sys, os
import time
import pandas as pd
import seaborn as sns
```

1.2.1 Start Pilot-Job

In [2]:
```
from pilot import PilotComputeService, ComputeDataService, State
COORDINATION_URL = "redis://EiFEvdHRy3mNBZDjsypraXGNQqJcAYKaTnHCZxgqLsykDoK

pilot_compute_service = PilotComputeService(coordination_url=COORDINATION_U

pilot_compute_description = {
                          "service_url": 'fork://localhost',
                          "number_of_processes": 1,
                         }

pilotjob = pilot_compute_service.create_pilot(pilot_compute_description=pil
```

```
Populating the interactive namespace from numpy and matplotlib
```

BigJob provides various introspection capabilities and allows the application to extract various details on the runtime.

In [8]:
```
pd.DataFrame(pilotjob.get_details().values(),
             index=pilotjob.get_details().keys(),
             columns=["Value"])
```

Out[8]:

	Value
bigjob_id	bigjob:bj-e758d79a-54a3-11e5-99b1-44a842265a41...
description	{'external_queue': 'PilotComputeServiceQueue-p...
start_time	1441549864.24
state	Running
stopped	False
nodes	['localhost\n']
end_queue_time	1441549867.93

In [9]:
```
compute_unit_description = {
        "executable": "/bin/sleep",
        "arguments": ["0"],
        "number_of_processes": 1,
        "output": "stdout.txt",
        "error": "stderr.txt",
}

compute_unit = pilotjob.submit_compute_unit(compute_unit_description)
compute_unit.wait()

# Print out some statistics about execution
pd.DataFrame(compute_unit.get_details().values(),
             index=compute_unit.get_details().keys(),
             columns=["Value"])
```

Out[9]:

	Value
run_host	radical-5
Executable	/bin/sleep
NumberOfProcesses	1
start_time	1441550025.18
agent_start_time	1441549867.93
state	Done
end_time	1441550028.33
Arguments	['0']
Error	stderr.txt
Output	stdout.txt
job-id	sj-47463332-54a4-11e5-99b1-44a842265a41
SPMDVariation	single
end_queue_time	1441550025.25

In [ ]:
```
pilot_compute_service.cancel()
```

2. Pilot-Hadoop

For the purpose of this tutorial we setup a Hadoop cluster on Chameleon (https://www.chameleoncloud.org/):

- YARN: http://129.114.108.119:8088/ (http://129.114.108.119:8088/)
- HDFS: http://129.114.108.123:50070/ (http://129.114.108.123:50070/)
- Ambari: http://129.114.108.119:8080/ (http://129.114.108.119:8080/)

2.1 Setup Spark on YARN

In [1]:
```
from numpy import array
from math import sqrt

%run env.py
%run util/init_spark.py

print "SPARK HOME: %s"%os.environ["SPARK_HOME"]

try:
    sc
except NameError:
    conf = SparkConf()
    conf.set("spark.num.executors", "4")
    conf.set("spark.executor.instances", "4")
    conf.set("spark.executor.memory", "5g")
    conf.set("spark.cores.max", "4")
    conf.setAppName("iPython Spark")
    conf.setMaster("yarn-client")
    sc = SparkContext(conf=conf)
    sqlCtx = SQLContext(sc)
```

```
SPARK HOME: /usr/hdp/2.3.0.0-2557/spark/
```

In [27]:
```
rdd = sc.parallelize(range(10))
```

In [28]:
```
rdd.map(lambda a: a*a).collect()
```

Out[28]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

3. KMeans

This is perhaps the best known database to be found in the pattern recognition literature. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant (see https://archive.ics.uci.edu/ml/datasets/Iris (https://archive.ics.uci.edu/ml/datasets/Iris)).

Source: R. A. Fisher, The Use of Multiple Measurements in Taxonomic Problems, 1936, http://rcs.chemometrics.ru/Tutorials/classification/Fisher.pdf (http://rcs.chemometrics.ru/Tutorials/classification/Fisher.pdf)

Pictures (Source Wikipedia (https://en.wikipedia.org/wiki/Iris_flower_data_set))

Setosa	Versicolor	Virginica

3.1 Load Data

In [6]:
```
data = pd.read_csv("https://raw.githubusercontent.com/pydata/pandas/master/
```

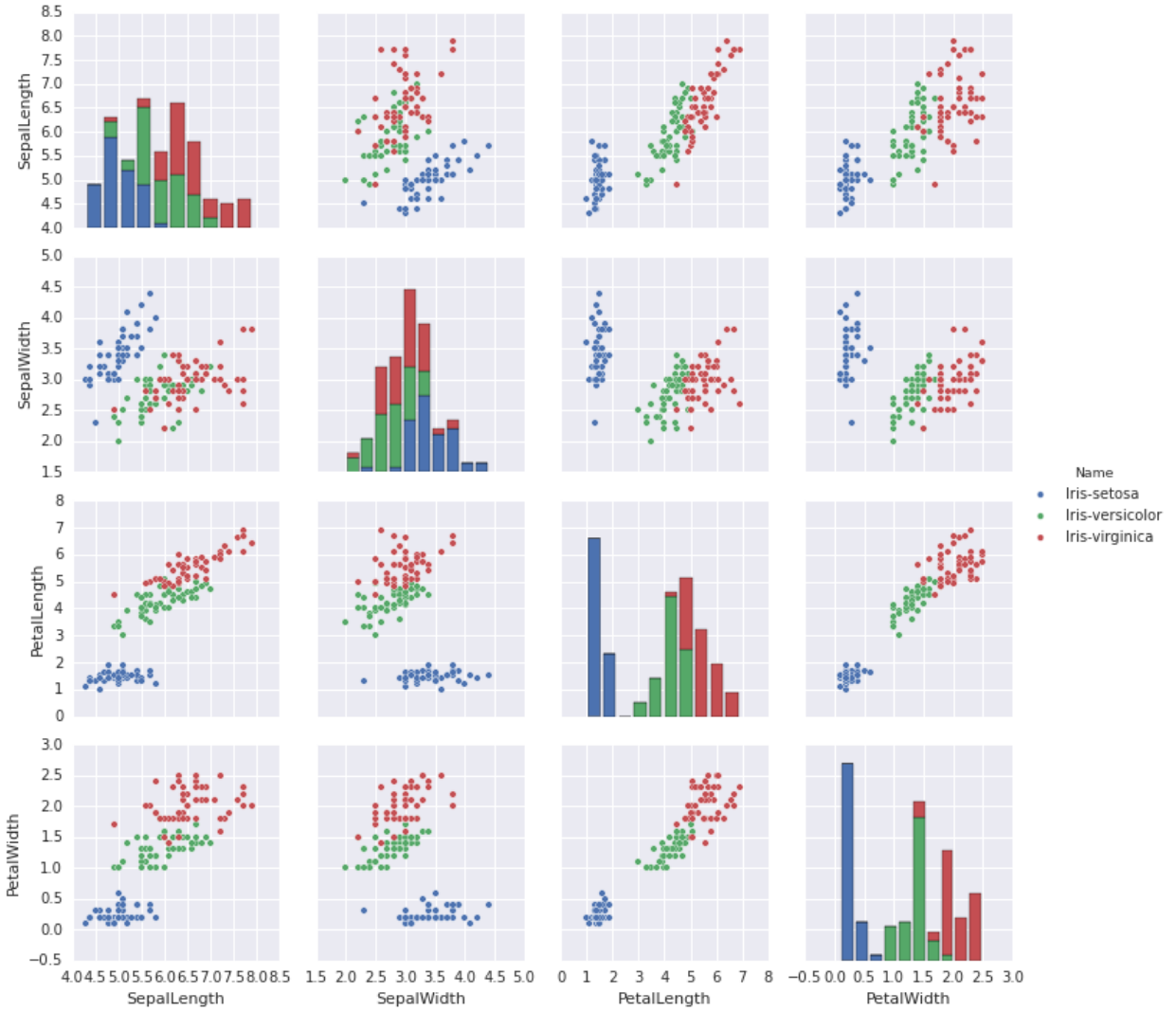
In [7]:
```
data.head()
```

Out[7]:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

The following pairplots show the scatter-plot between each of the four features. Clusters for the different species are indicated by the color.

In [4]:
```
sns.pairplot(data, vars=["SepalLength", "SepalWidth", "PetalLength", "Petal
```



3.2 KMeans (Scikit)

In [5]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3)
results = kmeans.fit_predict(data[['SepalLength', 'SepalWidth', 'PetalLengt
```

In [8]:
```
data_kmeans=pd.concat([data, pd.Series(results, name="ClusterId")], axis=1)
data_kmeans.head()
```

Out[8]:

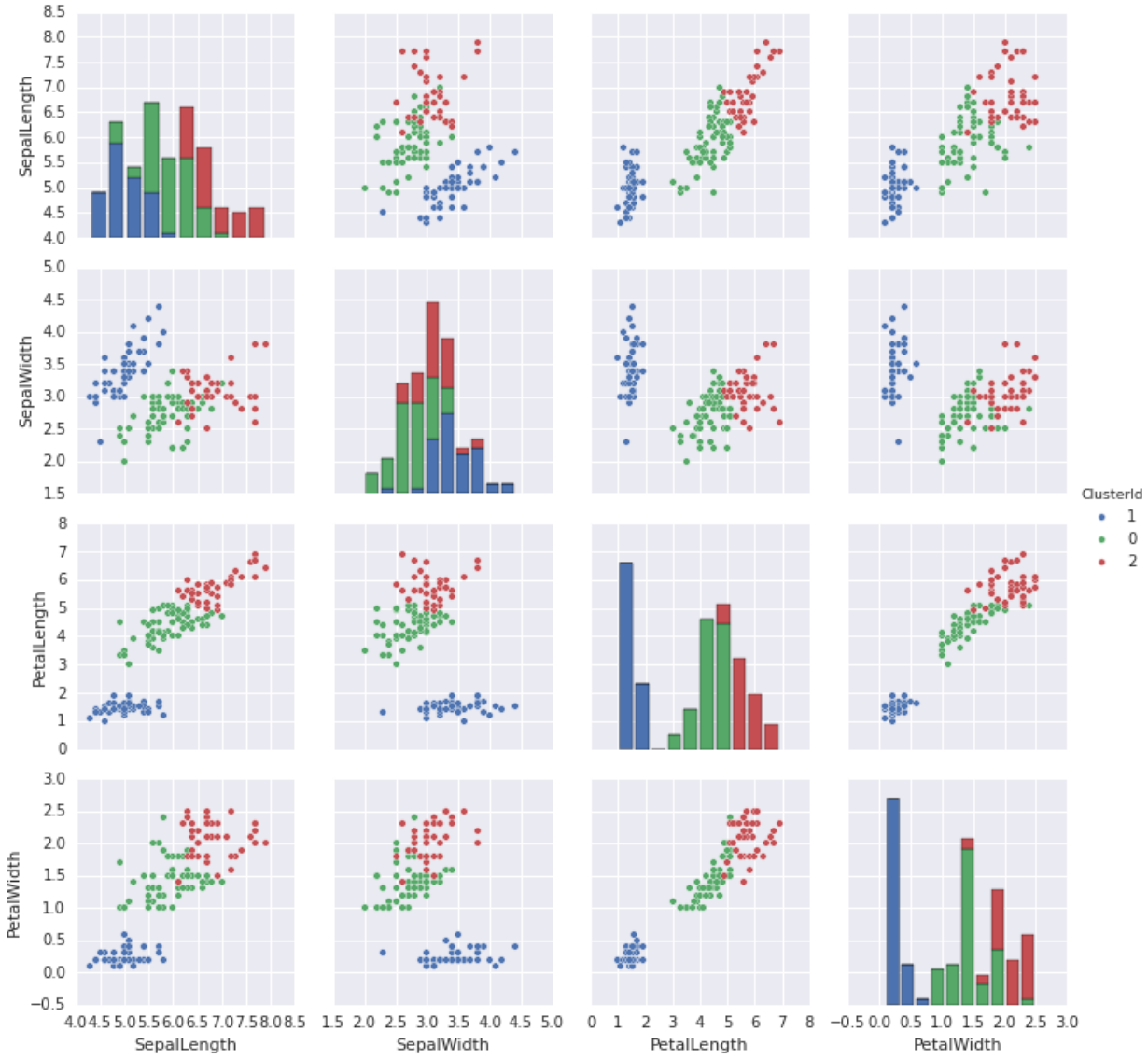
	SepalLength	SepalWidth	PetalLength	PetalWidth	Name	ClusterId
0	5.1	3.5	1.4	0.2	Iris-setosa	1
1	4.9	3.0	1.4	0.2	Iris-setosa	1
2	4.7	3.2	1.3	0.2	Iris-setosa	1
3	4.6	3.1	1.5	0.2	Iris-setosa	1
4	5.0	3.6	1.4	0.2	Iris-setosa	1

Evaluate Quality of Model

In [17]:
```
print "Sum of squared error: %.1f"%kmeans.inertia_
```

```
Sum of squared error: 78.9
```

In [12]:
```
sns.pairplot(data_kmeans, vars=["SepalLength", "SepalWidth", "PetalLength",
```



3.3 KMeans (Spark)

https://spark.apache.org/docs/latest/mllib-clustering.html#k-means (https://spark.apache.org/docs/latest/mllib-clustering.html#k-means)

In [8]:
```
data_spark=sqlCtx.createDataFrame(data)
```

In [16]:
```
data_spark_without_class=data_spark.select('SepalLength', 'SepalWidth', 'Pe
```

```
SepalLength SepalWidth PetalLength PetalWidth
5.1         3.5        1.4         0.2
4.9         3.0        1.4         0.2
4.7         3.2        1.3         0.2
4.6         3.1        1.5         0.2
5.0         3.6        1.4         0.2
5.4         3.9        1.7         0.4
4.6         3.4        1.4         0.3
5.0         3.4        1.5         0.2
4.4         2.9        1.4         0.2
4.9         3.1        1.5         0.1
5.4         3.7        1.5         0.2
4.8         3.4        1.6         0.2
4.8         3.0        1.4         0.1
4.3         3.0        1.1         0.1
5.8         4.0        1.2         0.2
5.7         4.4        1.5         0.4
5.4         3.9        1.3         0.4
5.1         3.5        1.4         0.3
5.7         3.8        1.7         0.3
5.1         3.8        1.5         0.3
```

Convert DataFrame to Tuple for MLlib

In [30]:
```
data_spark_tuple = data_spark.map(lambda a: (a[0],a[1],a[2],a[3]))
```

Run MLlib KMeans

In [31]:
```
# Build the model (cluster the data)
from pyspark.mllib.clustering import KMeans, KMeansModel
clusters = KMeans.train(data_spark_tuple, 3, maxIterations=10,
                        runs=10, initializationMode="random")
```

Evaluate Model

In [34]:
```
# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = data_spark_tuple.map(lambda point: error(point)).reduce(lambda x, y
print("Within Set Sum of Squared Error = " + str(WSSSE))
```

```
Within Set Sum of Squared Error = 97.3259242343
```

4. Graph Analysis

4.1 Load Data

In [43]:
```
import networkx as NX
```

In [38]:
```
graph_data = pd.read_csv("https://raw.githubusercontent.com/drelu/Pilot-KMe
                         names=["Source", "Destination"])
```

In [39]:
```
graph_data.head()
```

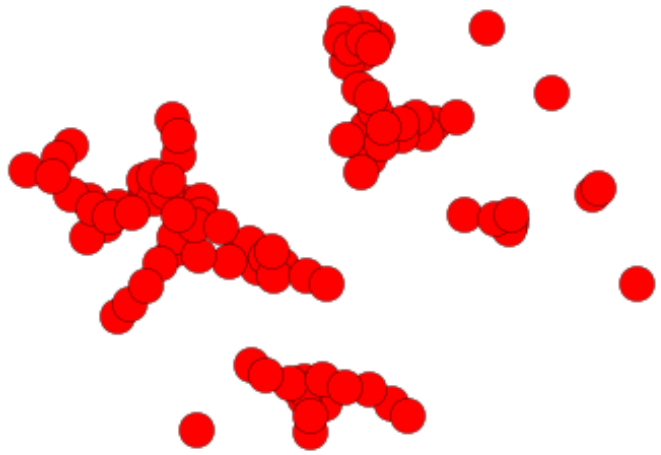
Out[39]:

	Source	Destination
0	0	0
1	0	67
2	0	14
3	1	1
4	1	41

In [53]:
```
nxg = NX.from_edgelist(list(graph_data.to_records(index=False)))
```

4.2 Plot Graph

In [54]:
```
NX.draw(nxg, pos=NX.spring_layout(nxg))
```

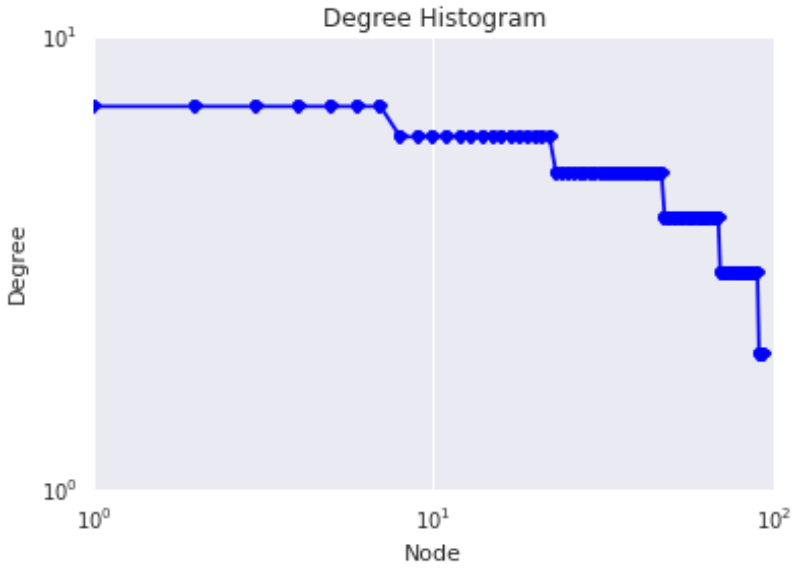
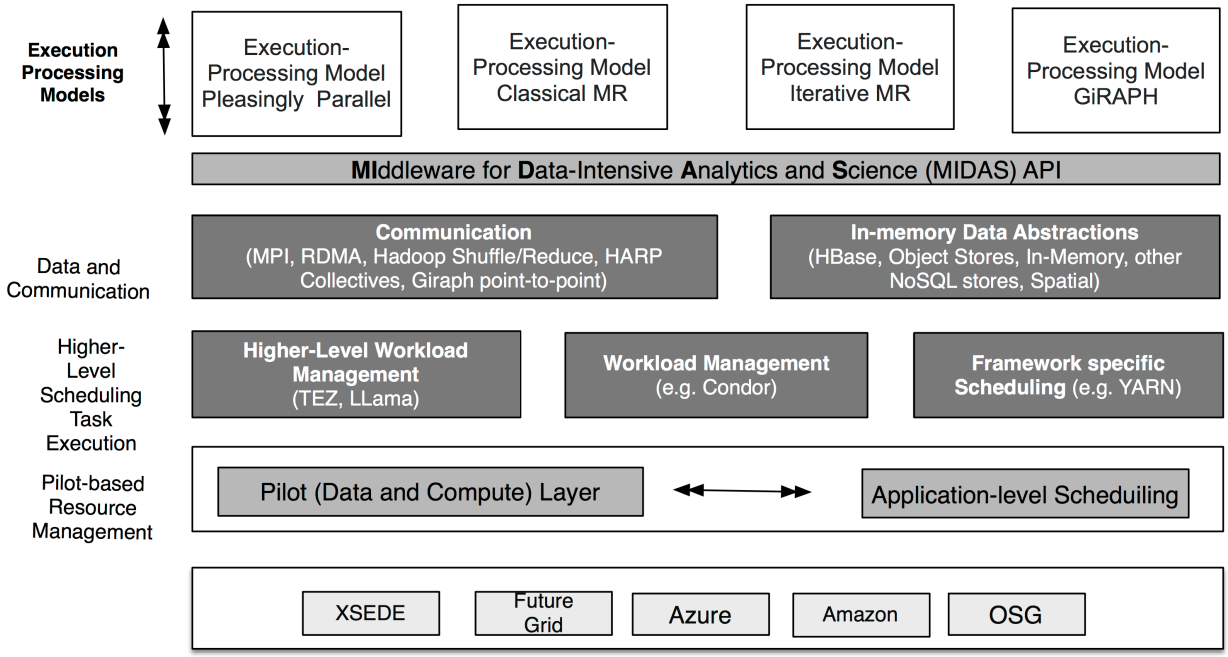


4.3 Analytics

Degree Histogram

In [52]:
```
import matplotlib.pyplot as plt
degree_sequence=sorted(NX.degree(nxg).values(),reverse=True) # degree seque
#print "Degree sequence", degree_sequence
#print "Length: %d" % len(degree_sequence)
dmax=max(degree_sequence)
plt.loglog(degree_sequence,'b-',marker='o')
plt.title("Degree Histogram")
plt.ylabel("Degree")
plt.xlabel("Node")
```

Out[52]: <matplotlib.text.Text at 0x7f7945745710>



5. Future Work: Midas