ユーザへの処理透過性を実現する
スケールアウト可能な
分散データ処理基盤に関する研究

2016/02/19

IIJ技術研究所 阿部 博(abe@iij.ad.jp)

自己紹介

- 所属
  - IIJ技術研究所 研究員
  - WIDEプロジェクトメンバー
  - 北陸先端科学技術大学院大学 篠田研究室 博士後期課程学生
- 経歴
  - IIJ: 大規模データ処理/クラウド(IaaS/PaaS)に関する業務を経験
  - IIJ技術研究所: データセンター/分散処理に関する研究
  - Interop NOCメンバー: モニタリング、データ収集/可視化

発表内容

- 背景
- 透過的な環境
- 分散環境におけるプログラミング
- Work in progress
- まとめ

背景

Data volume in co-IZmo SD

- Rough estimate of data entries (entry / 5 sec)

	1 device	1 Rack	1 co-IZmo SD(3 Rack)
PDU	100	200(2 PDU)	600
Switch	500	1,500(3 switches)	4,500
Server	100	4,000(40 servers)	12,000
others(UPS, iSCSI, sensors…)	1,000	1,000	3,000
Total	1,700	5,700	21,000

  - co-IZmo SD's data entries per sec = about 4k
    - 14M entries /1 hour, 345M entries / 1 day, 126G entries / 1 year
- Data size
  - 1 entry = 32 byte(time, value format)
  - 126G entries x 32 byte = 4TByte / year

松江DCP規模のデータサイズ

- 3ラックで4TB/年(co-IZmo SD試算)
- 松江DCPでの稼働コンテナ数(32コンテナ)
  - 1コンテナ＝9ラック
  - 32コンテナ x 9ラック = 288ラック
- 3ラック：4TB = 288ラック：384TB(なかなかにBigData)

BigData

- ユーザが扱うデータ量の増大
  - 数GBから数TB、数百TBから数PBの世界へ
- データの種類
  - ストアドデータ、ストリームデータ
- 処理の内容
  - 静的解析、傾向分析、アノマリ検知

処理方法

- バッチ処理
  - 大きなデータからマイニング
- ストリーム処理
  - 少量のデータ処理/短時間のマイクロバッチ処理

計算機1台での処理限界

- リソースが足りない
  - メモリ/CPU cores/ストレージ
- 時間をかければ処理可能な事もある
  - 数時間/数十時間
- 分散処理システム/フレームワークを使って限界を打破
  - Hadoopエコシステム、クラウドソリューション

一般解

- co-IZmo SDのデータ解析用クラスタ
- データ処理ソフトウェア
  - MapReduce, Hive, Impala
- クラスタ管理ソフトウェア
  - YARN
- 分散ストレージ
  - HDFS

ハードウェアスペック(FUJITSU RX200/S8)

- CPU Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz x 2
- 12 core(HT 24 core)
- メモリ128GB
- ディスク(SSD 200GB x 4, PCI SSD 200GB x 2) 合計 1.2TB

分散処理システムは難しい

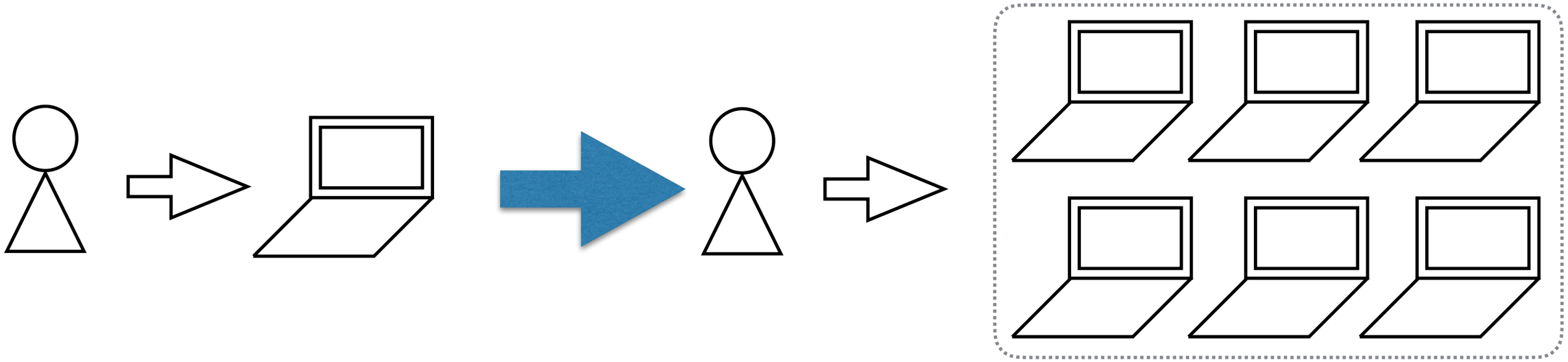
- 複数台の計算機をクラスタリングして利用
  - 構築/運用の難しさ/チューニングの難しさ
- 複雑なプログラミングモデル/データフォーマット
  - 多種多様なフレームワークの利用/専用手法の学習コスト
- 効率的なリソース利用
  - バランスのとれたCPU/メモリ/ストレージの利用/配置

学習コスト

- 運用
  - Spark, Impala, Hive, YARN, HDFS, …
- データ処理手法
  - MapReduce, Hive(HiveQL), Spark(Spark SQL, PySpark), Impala(SQL)
- データフォーマット
  - Parquet, RDD, Kudu, …

エンドユーザが欲しいもの

- ローカル環境と分散環境で同様の使用感(透過性)
  - 変わらないインタフェース
  - 変わらないバックエンドの利用

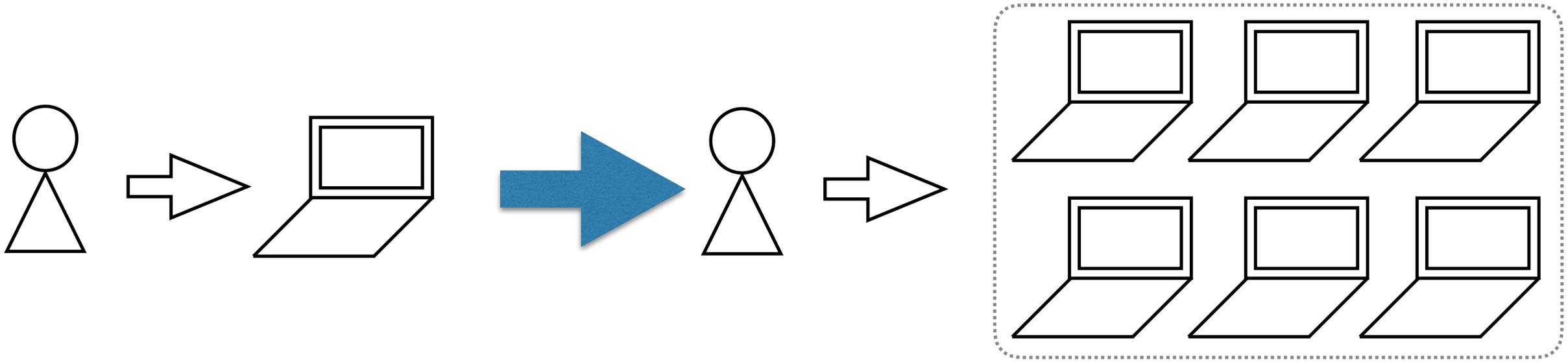


ローカル環境　　分散環境

~~エンドユーザ~~が欲しいもの

僕

- ローカル環境と分散環境で同様の使用感(透過性)
  - 変わらないインタフェース
  - 変わらないバックエンドの利用



ローカル環境　　分散環境

ゴール

- データへのアクセス/システム利用の透過性、分散システム/プログラミング、運用負担の軽減
  - 「同時に成り立たせるのが難しいものを成り立たせる」
- お金で解決できる部分（エンジニアリング）
  - 運用負担の軽減
- お金で解決できない部分
  - 透過的な分散システムの実現、分散処理プログラミング

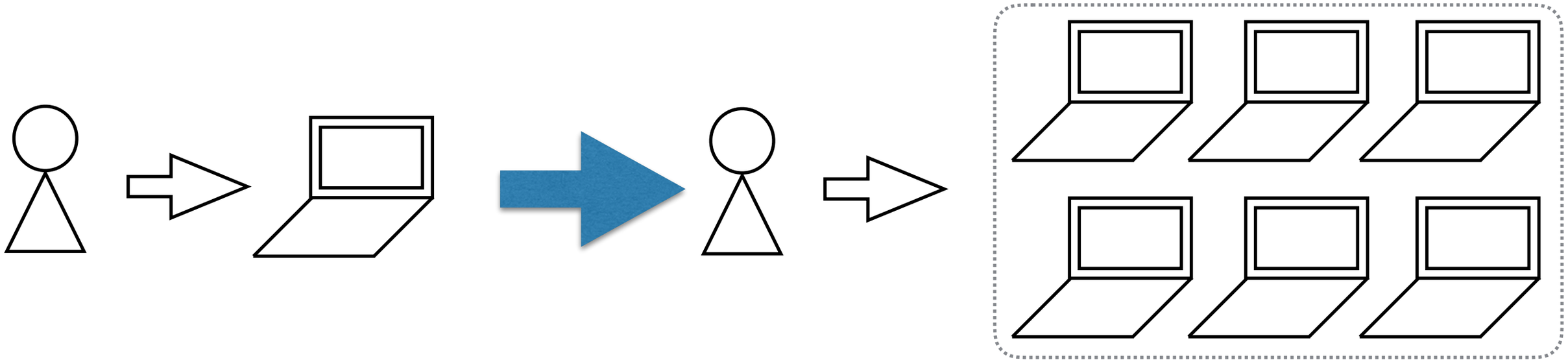
必要な要件

- 透過的分散システムアーキテクチャ
  - 分散処理システムへの透過性
  - ストレージアクセスへの透過性
- 分散処理システム向けプログラミング
  - 処理手法の共通化
  - フレームワークを意識させない抽象化層の実現

透過的な環境

エンドユーザが欲しいもの(再掲)

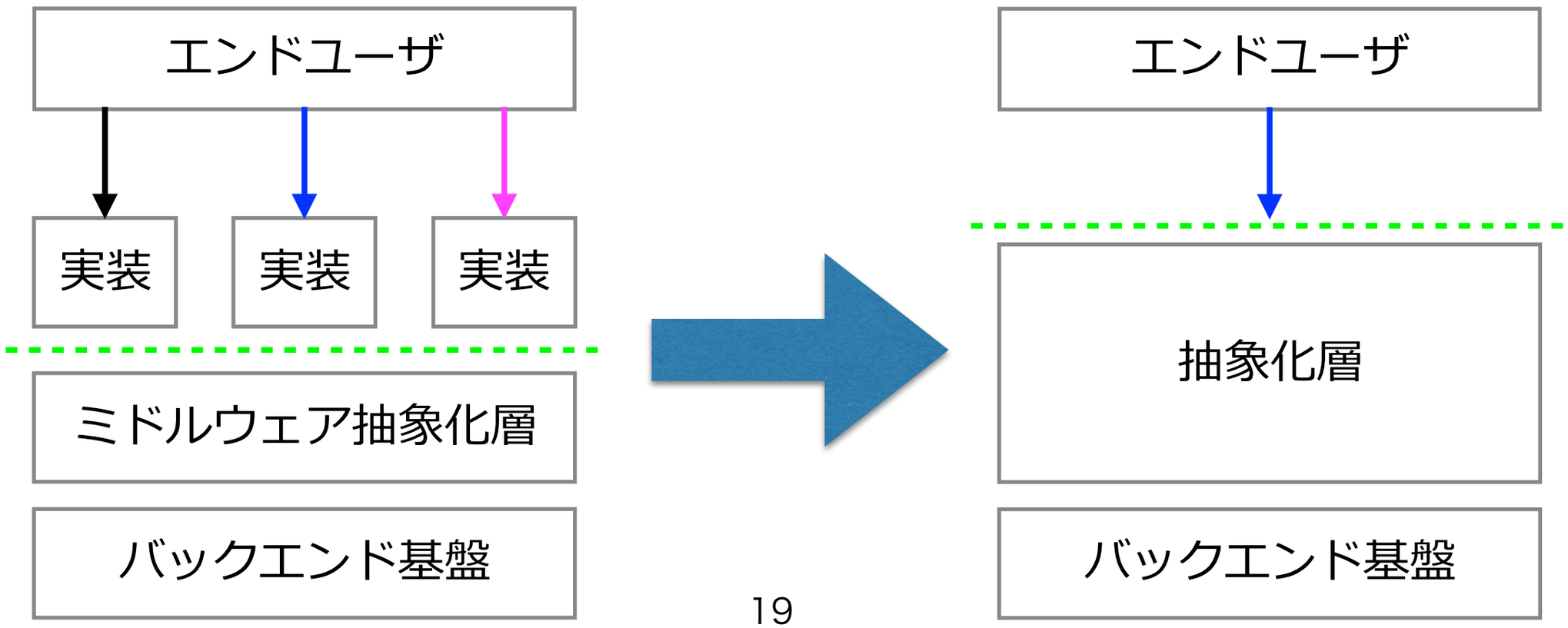
- ローカル環境と分散環境で同様の使用感(透過性)
  - 変わらないインタフェース
  - 変わらないバックエンドの利用



バックエンドを隠蔽

- エンドユーザからの要求とバックエンドでの処理を橋渡しする抽象化層

- バックエンドを隠蔽するミドルウェア



ストレージ透過性

- 分散ファイルシステムへのアクセス
  - ファイル/ディレクトリへのアクセス透過性を担保
- ファイルに対するread/write処理(raw)
  - csv, tvs, …
- 特定のデータ形式に対する処理(raw -> structured)
  - array, hash, DataFrame

ローカル環境と同じ処理言語

- Python
  - ライブラリ: NumPy, Pandas, SciPy, Matplotlib
  - フォーマット: csv, tsv
  - データ形式: array, narray, DataFrame
- Ruby
  - …

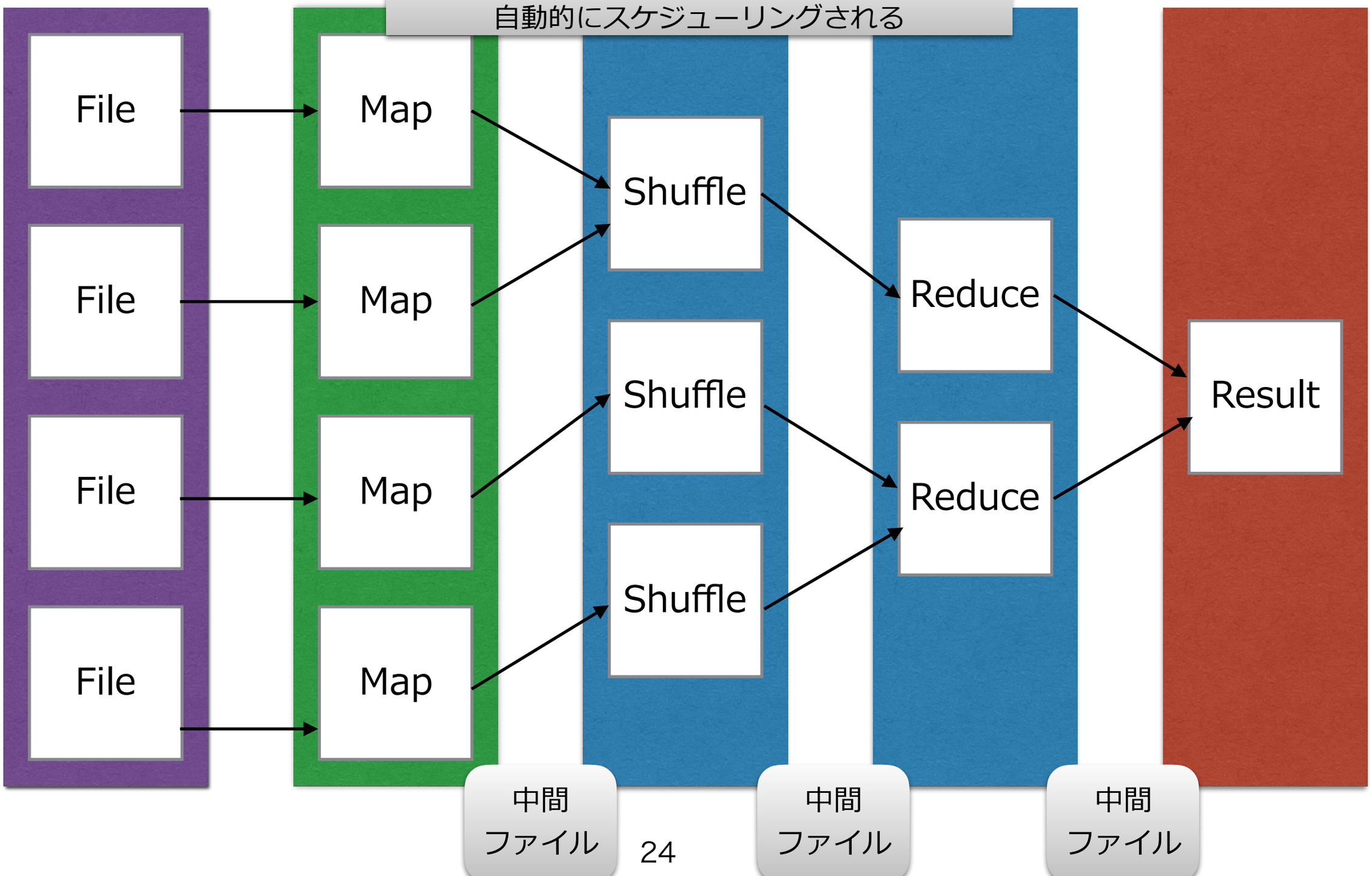
分散環境における
プログラミング

処理の実行スケジューリング

- $ cat *.csv | python hoge.py | python fuga.py > result
  - 分散実行されない/順序保証されない/並列実行されない
- $ xargs --max-procs=xを使う方法
  - 並列実行はされる/結果の同期は行われない
- 分散実行結果の同期
  - タスク分解と分解したタスクのスケジューリングが必要

MapReduceの例

MapperとReducerさえ記述すれば
自動的にスケジューリングされる



DAG(Directed Acyclic Graph)

- MapReduceを意識しないプログラミングの実現
- 有効非巡回グラフの利用
- 処理を複数のプロセスへ分解
- スケジューラで複数プロセスへの処理を制御

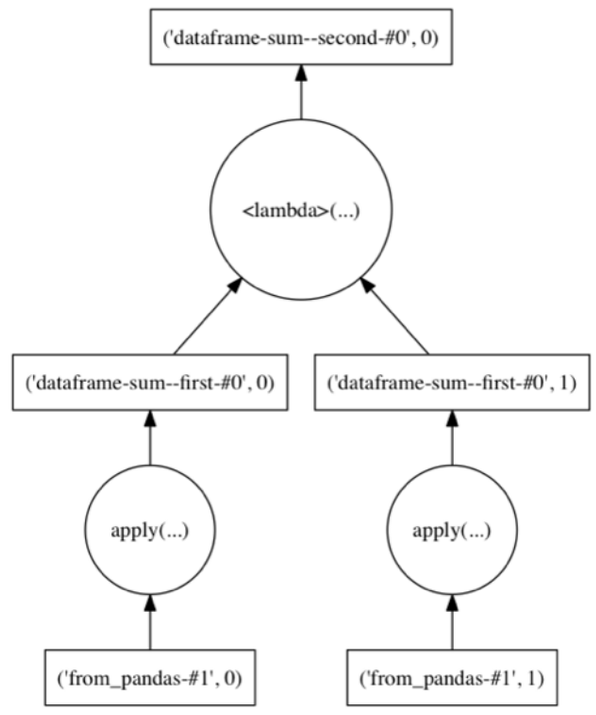
```
In [4]: # %load load_dask.py
        import numpy as np
        import pandas as pd
        import dask.dataframe as dd

        df = pd.DataFrame({'X': np.arange(10),
                           'Y': np.arange(10, 20),
                           'Z': np.arange(20, 30)},
                          index=list('abcdefghij'))

        ddf = dd.from_pandas(df, 2)

In [5]: ddf.sum().compute()
Out[5]: X     45
        Y    145
        Z    245
        dtype: int64

In [6]: ddf.sum().visualize()
Out[6]:
```

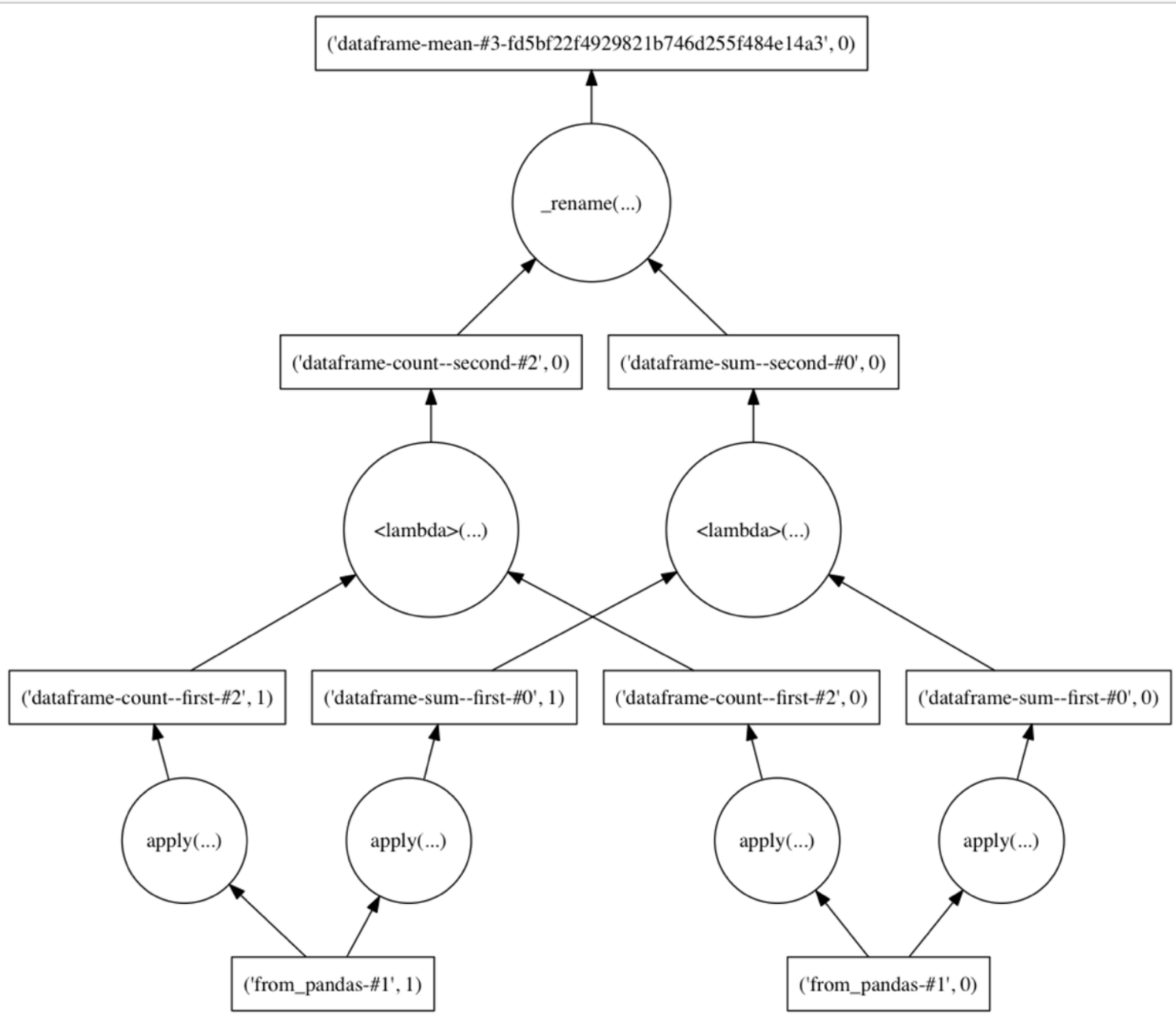


DAG(Directed Acyclic Graph)

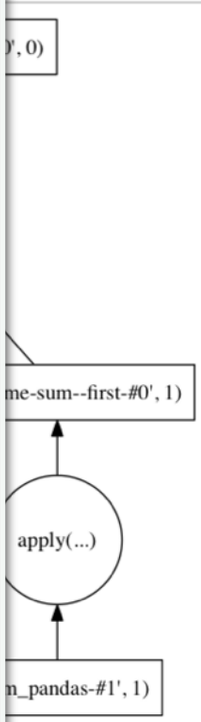
- Map
  プロ
- 有効
- 処理
- スケ
  の処

```
In [15]: ddf.mean().compute()
Out[15]: X     4.5
         Y    14.5
         Z    24.5
         dtype: float64

In [16]: ddf.mean().visualize()
Out[16]:
```




```
range(10),
range(10, 20),
range(20, 30)},
('abcdefghij'))
```



分散処理に効果的な技法

- マルチプロセス/マルチスレッド
  - 処理の順序/同期の制御
- Out-of-Core
  - 実メモリに乗らないデータをディスクから読みつつ処理
- ファイル複製/分割
  - ノード複製による耐障害性/ノードが持つファイルへの処理

必要な要件

- 透過的分散システムアーキテクチャ
  - 分散処理システムへの透過性
  - ストレージアクセスへの透過性
- 分散処理システム向けプログラミング
  - 処理手法の共通化
  - フレームワークを意識させない抽象化層の実現

もう一つの必要な要件

必要な要件

- 透過的分散システムアーキテクチャ
  - 分散処理システムへの透過性
  - ストレージアクセスへの透過性
- 分散処理システム向けプログラミング
  - 処理手法の共通化
  - フレームワークを意識させない抽象化層の実現

「2つの要件を満たしつつ
高速に動作する」

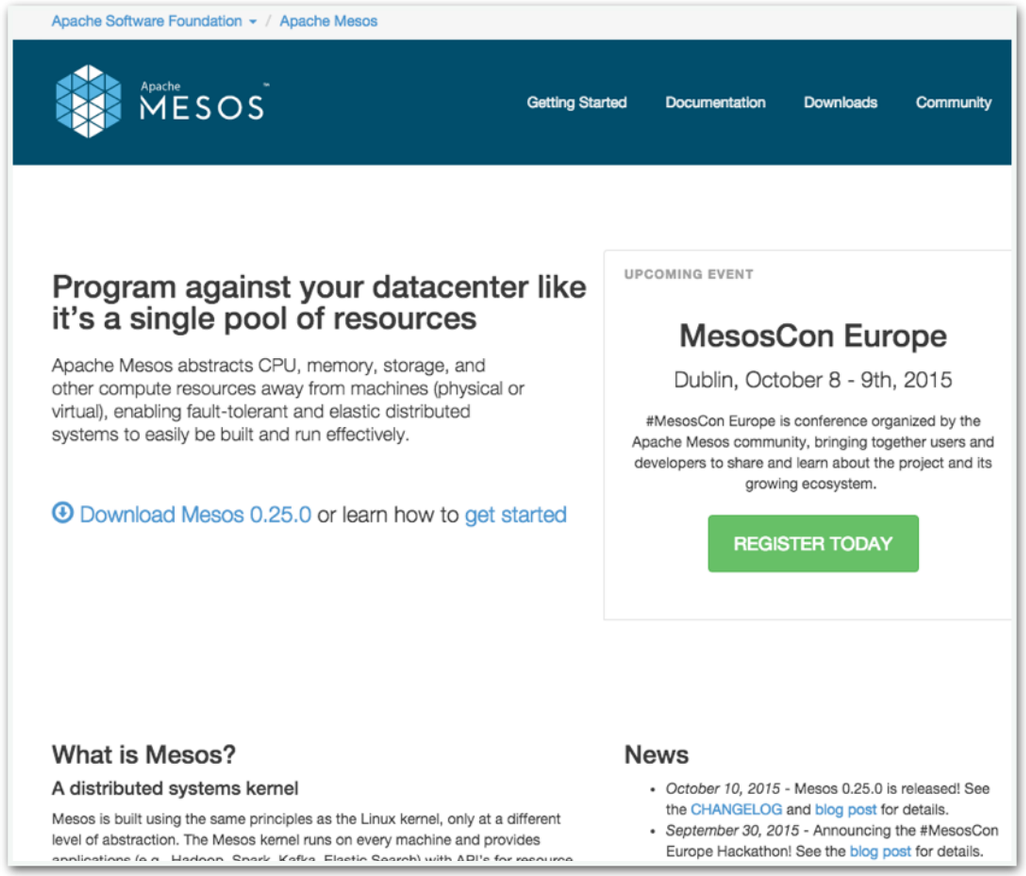
Work in progress

Proof of Concept(ver 0.1)

- Mesosを用いた分散システムの隠蔽
- 分散ストレージへのアクセス抽象化

Apache Mesosとは？

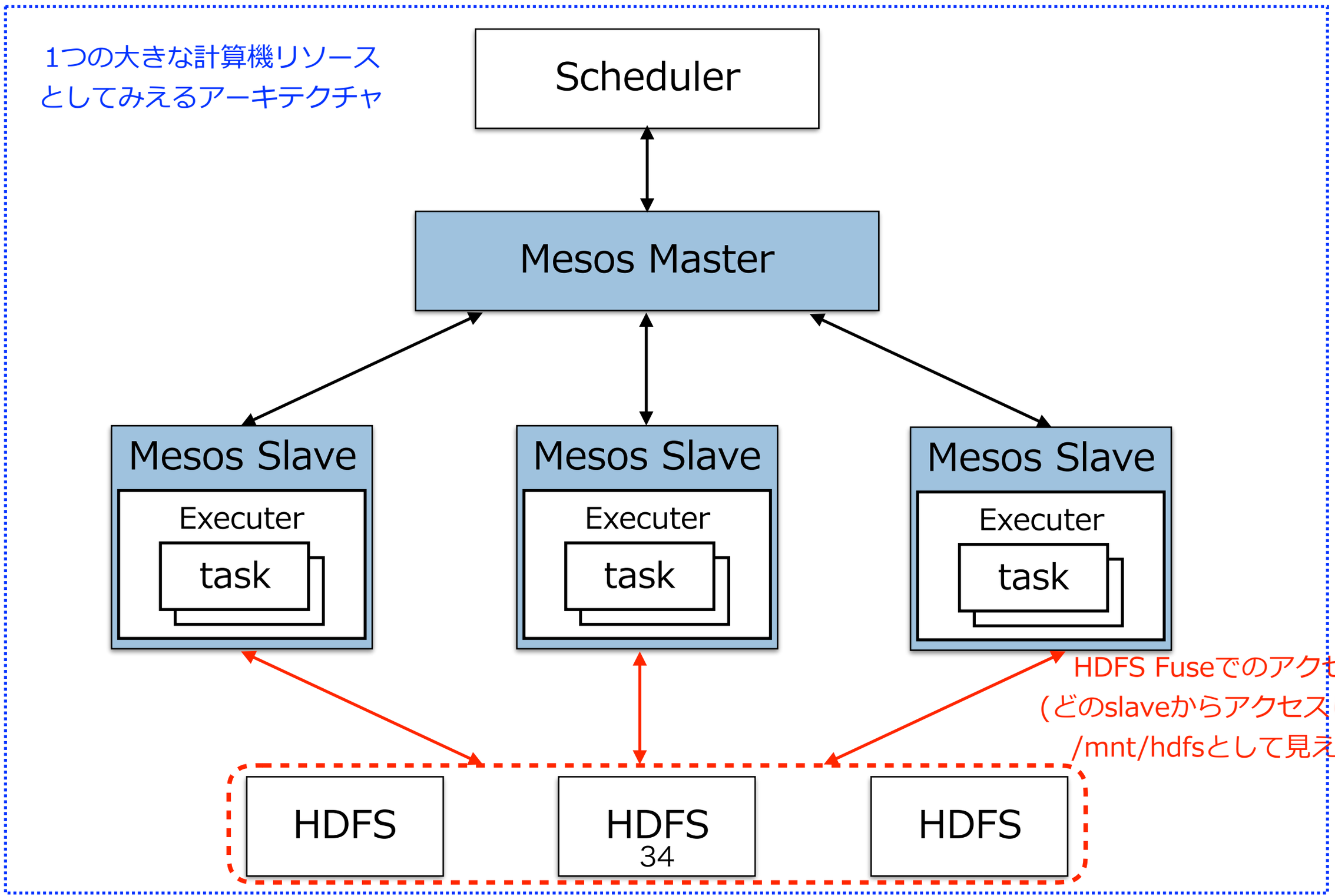
- http://mesos.apache.org/
  - Apache Projectの一つ
- A distributed systems kernel
  - DCOS(DataCenter OS)
- Mesosでできること
  - Zookeeperを利用したMasterの高信頼性の実現
  - Slaveノードの追加によるスケールアウトの実現
  - 複数フレームワーク同時実行による利用率の向上



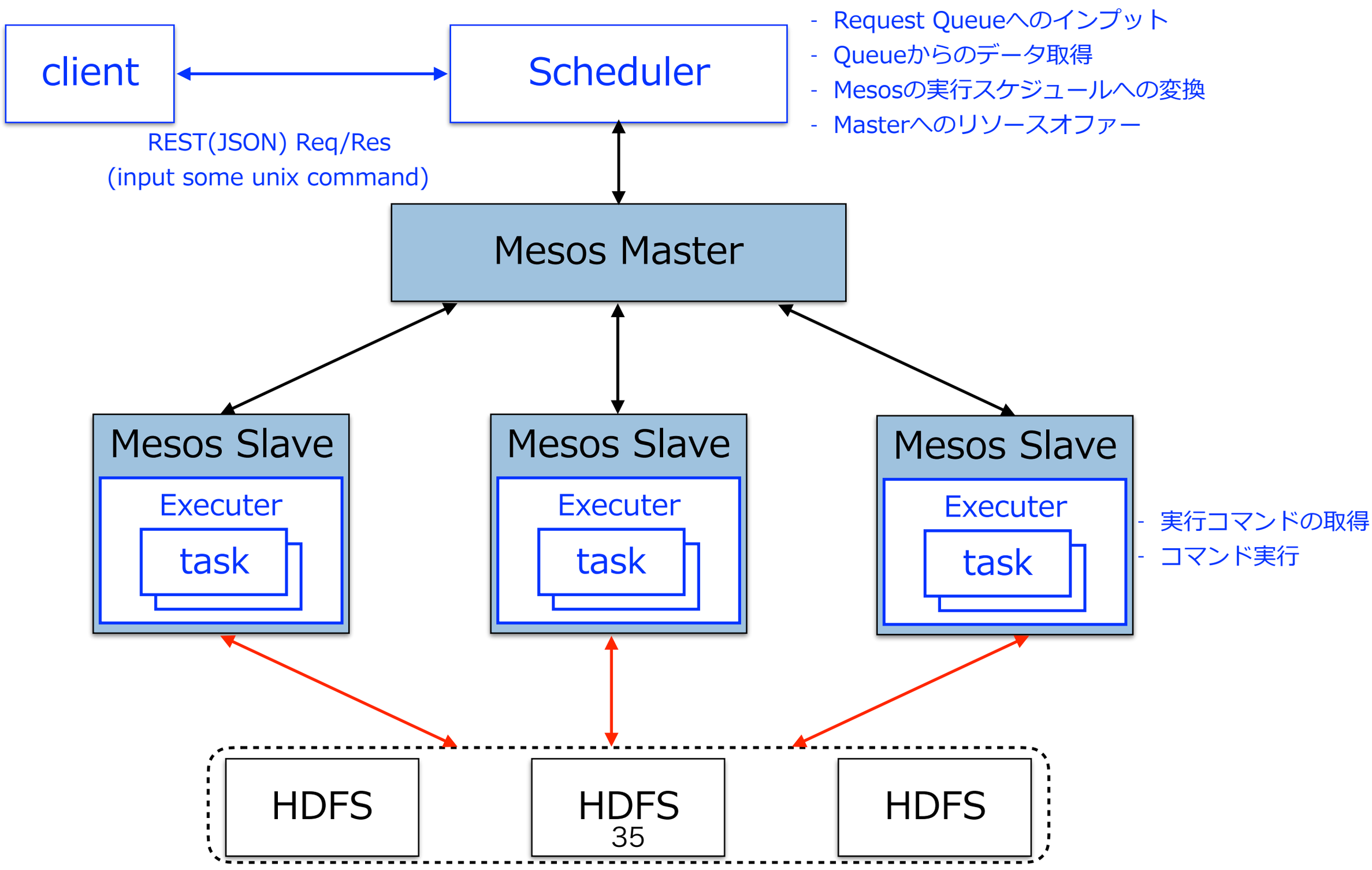
フレームワークのための抽象化層

- Mesosを使うメリット
  - フレームワーク(Hadoop,Spark,Docker, …)間差異を吸収
  - リソースコントロール/障害時フェイルオーバーの委譲
  - スケールアウトの容易性(Slaveを足すだけ)

透過性の実現


1つの大きな計算機リソース
としてみえるアーキテクチャ
Scheduler
Mesos Master
Mesos Slave
Executer
task
Mesos Slave
Executer
task
Mesos Slave
Executer
task
HDFS Fuseでのアクセス
(どのslaveからアクセスしても
/mnt/hdfsとして見える)
HDFS
HDFS
HDFS

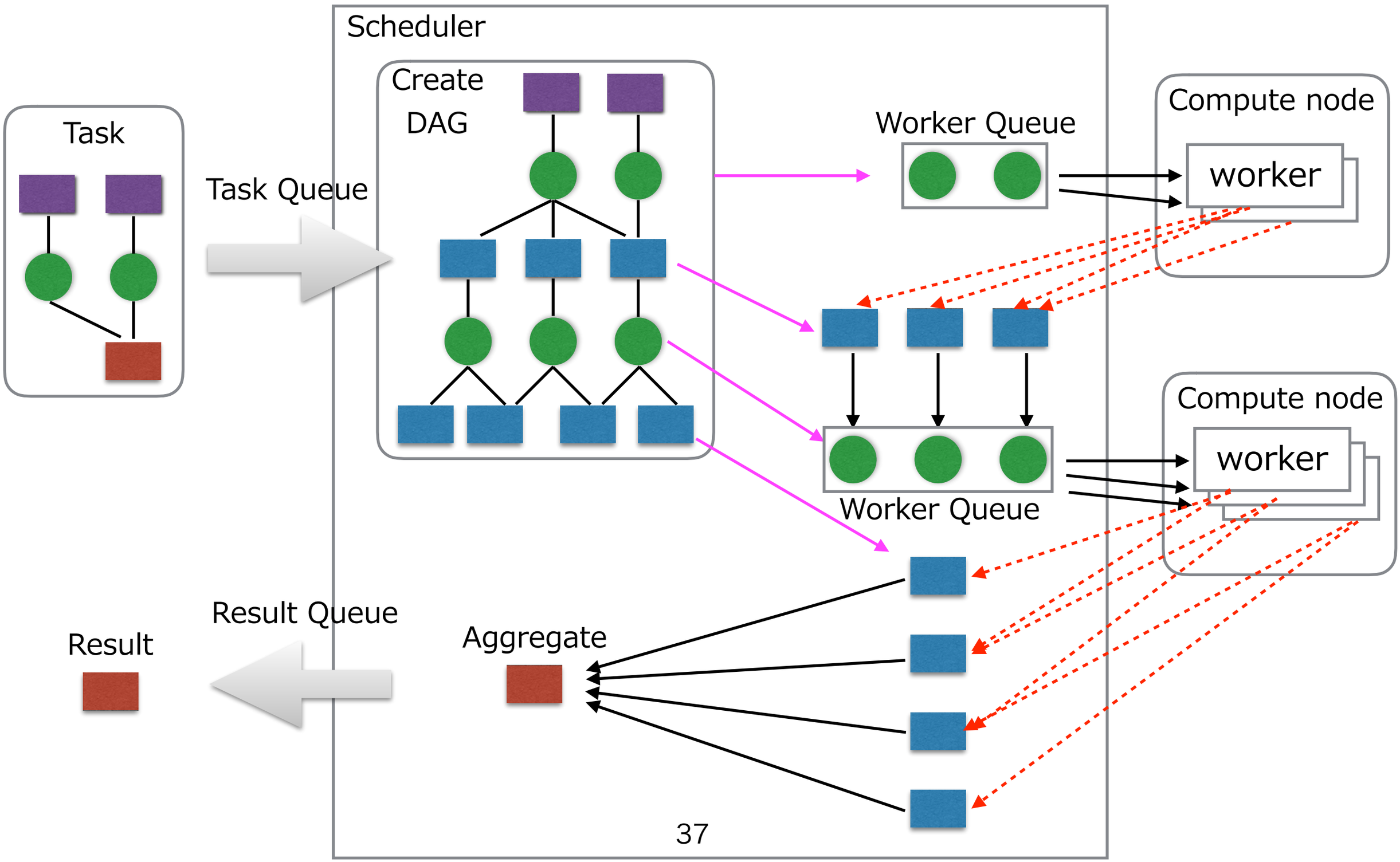
独自フレームワークの作成



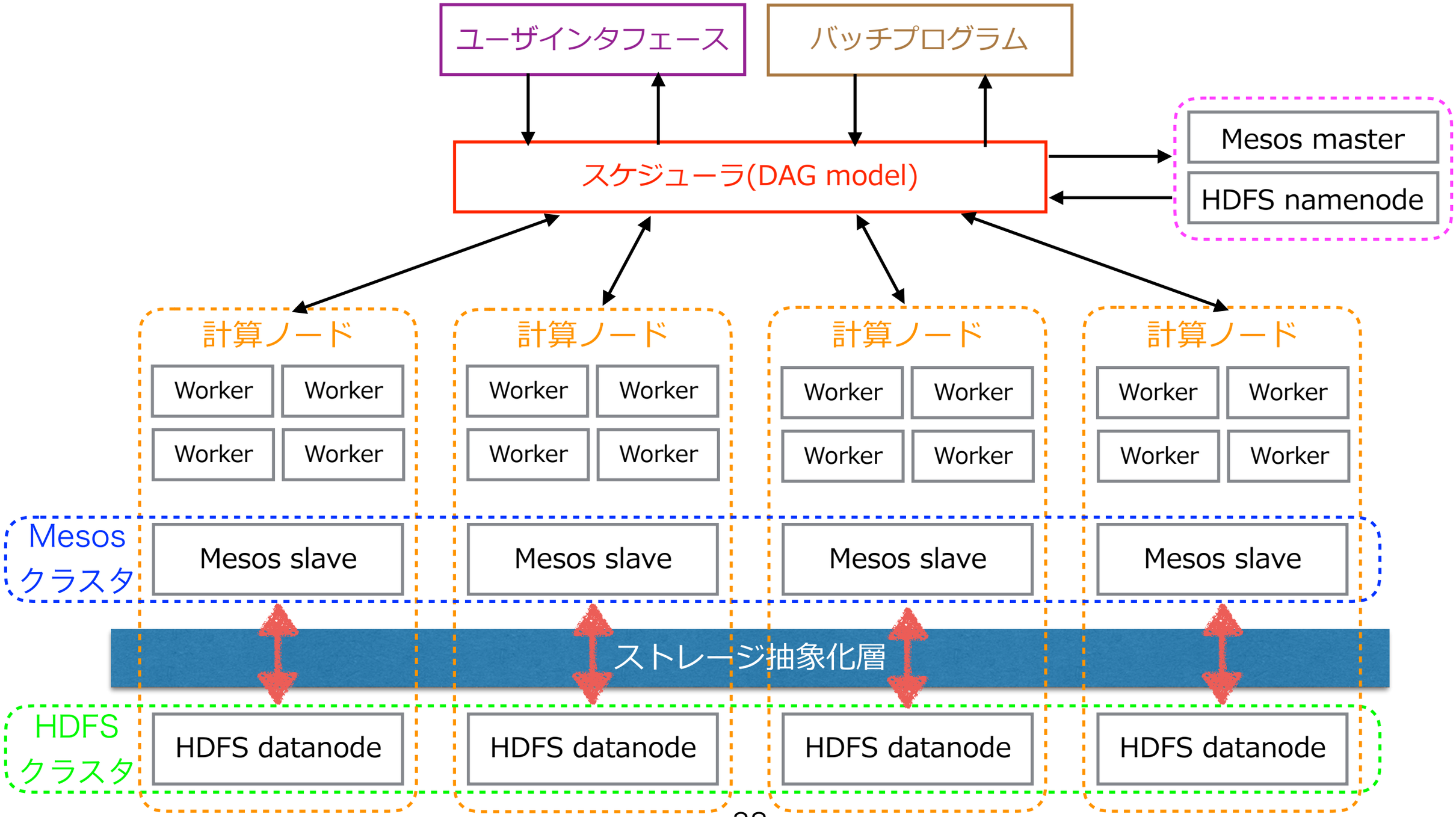
Proof of Concept(ver 0.2)

- プログラミングモデル(DAG)の実現
- DAGと連携したスケジューラの実装
- HDFSの性能を引き出すFUSEではない透過性実現

DAGを使った分散実行フロー

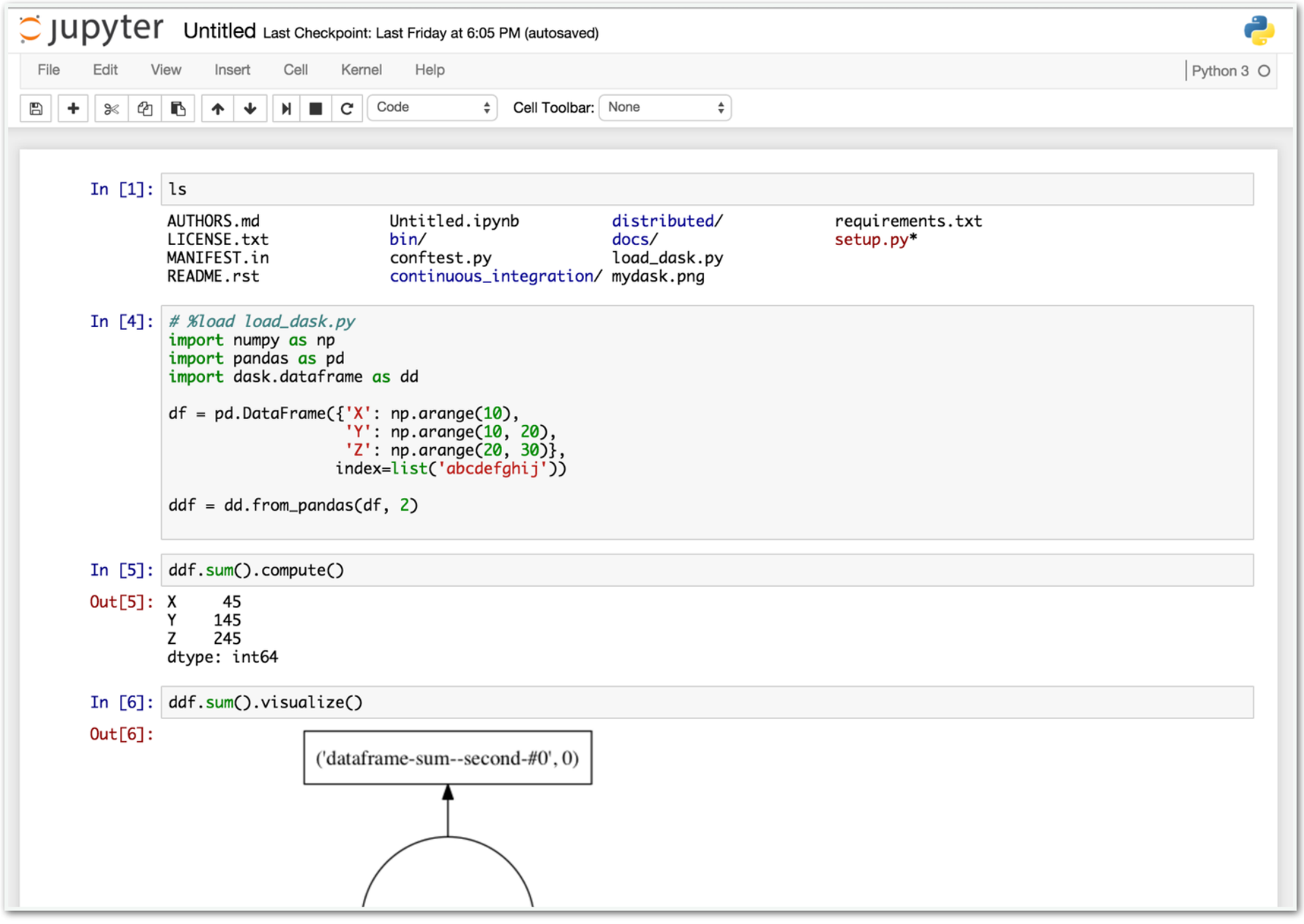


システムアーキテクチャ



ユーザインタフェース

- jupyter notebook(iPython notebook)



処理の高速化

- データアクセスの高速化
  - HDD -> SSD、メモリをたくさん積む
- GPGPU
  - PyCudaを使った行列演算の底上げ
- Numba
  - PythonのJITコンパイラによる高速化

言語

- Python
  - NumPy, SciPy, Pandasなどを透過的に利用
  - Numbaを使った高速化
- Golang
  - gorutine/channelによる並行処理の隠蔽
  - バイナリの実行環境

まとめ

- 分散システムで透過性を実現するときの課題
- 問題解決へのシステム設計とプログラミングモデル
- PoCの現状と今後

- システムの名前つけなきゃ…