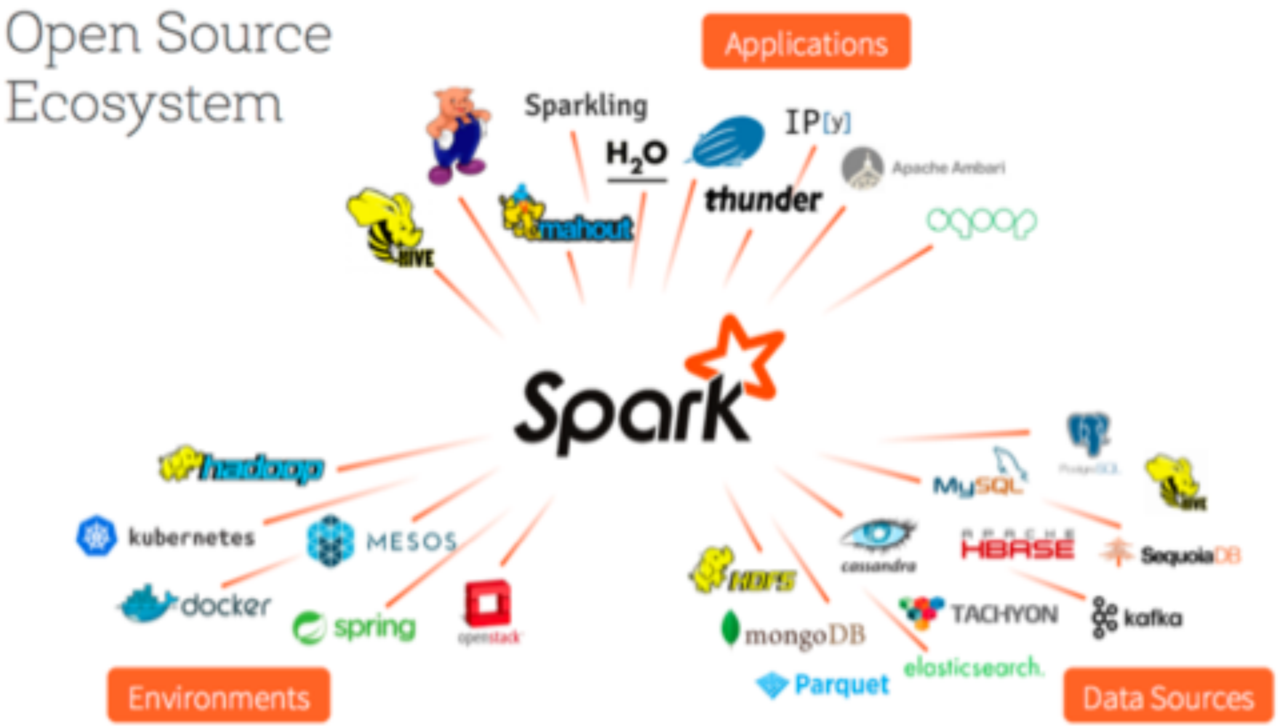
Serverless Data Analytics with Flint

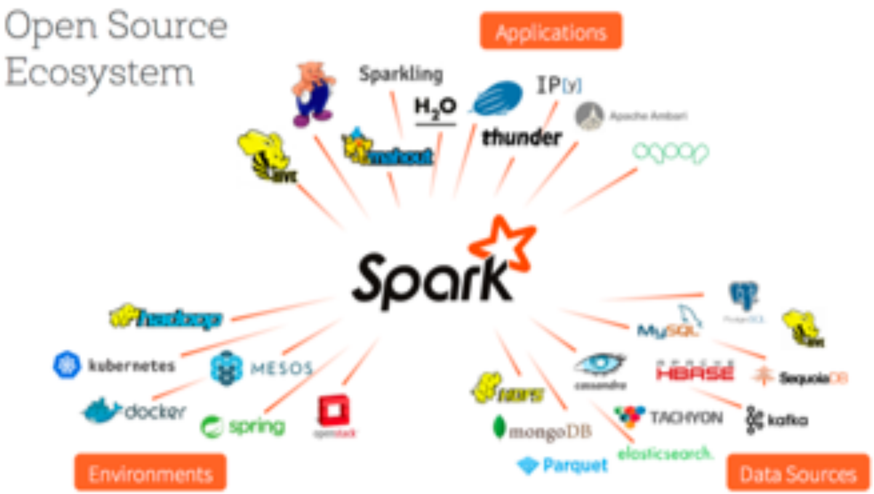
YOUNGBIN KIM AND JIMMY LIN

Large-scale analytical data processing



- Spark adoption is booming
- Many use cases

Large-scale analytical data processing



- Spark adoption is booming
- Many use cases

=> Requirement: Pre-installed on a cluster before they can be used for analytics

- On-premise data center or cluster of virtual instances in the cloud

Large-scale analytical data processing

- Problem: Cluster management can be difficult
  - monitoring the health of worker nodes
  - troubleshoot a variety of issues
  - fixing/replacing underperforming nodes

May not be feasible for many small startups/researchers with the limited resources !

Large-scale analytical data processing

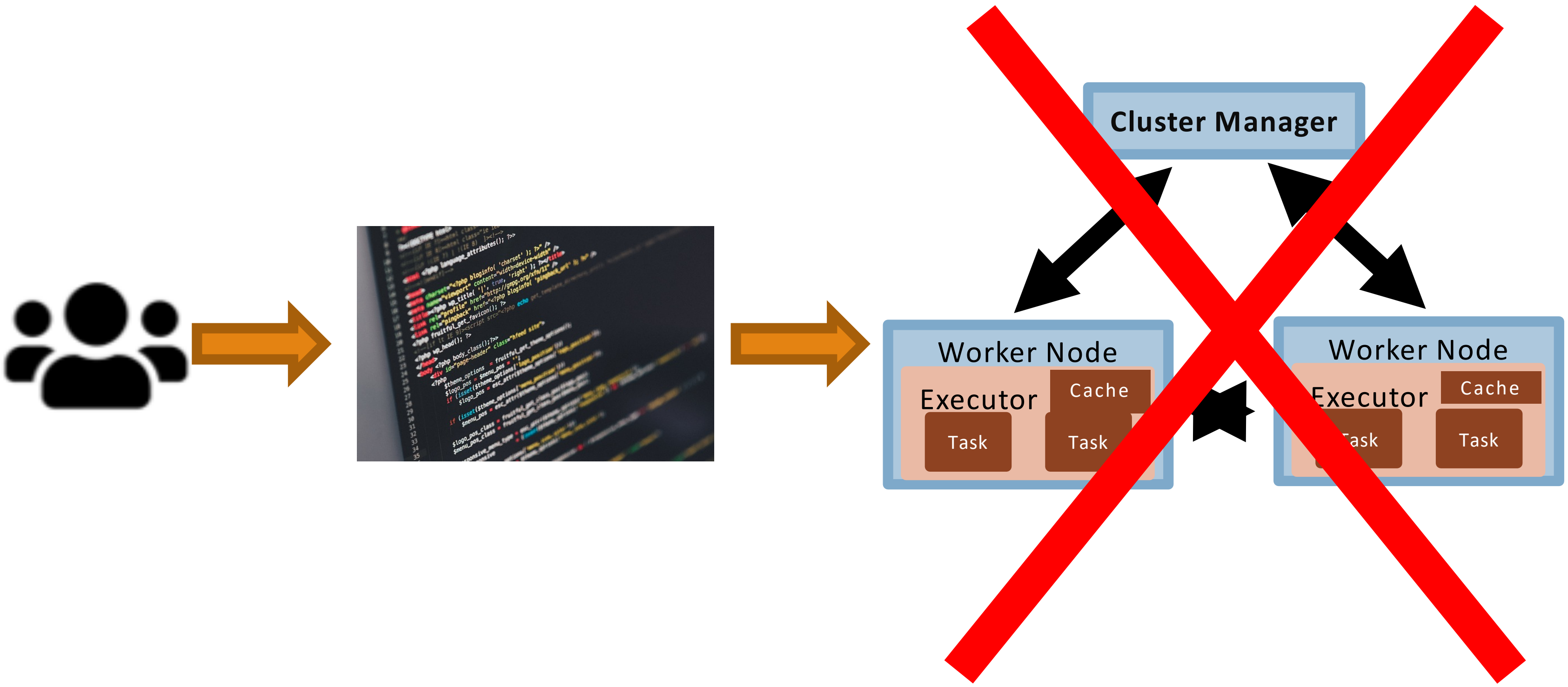
- How about scaling?

Managed big data frameworks

- Current solution: Managed big data frameworks
- Example: Amazon Elastic Map Reduce (EMR)
- Advantages:
  - Reduces the burden of cluster management
  - Save costs (automatically terminated)
- Limitations:
  - Time is wasted in cluster initialization/rescaling/teardown
  - Need to choose the details of the managed cluster
- There are still management overheads & idle costs.

Serverless analytics

- Serverless analytics to the rescue!



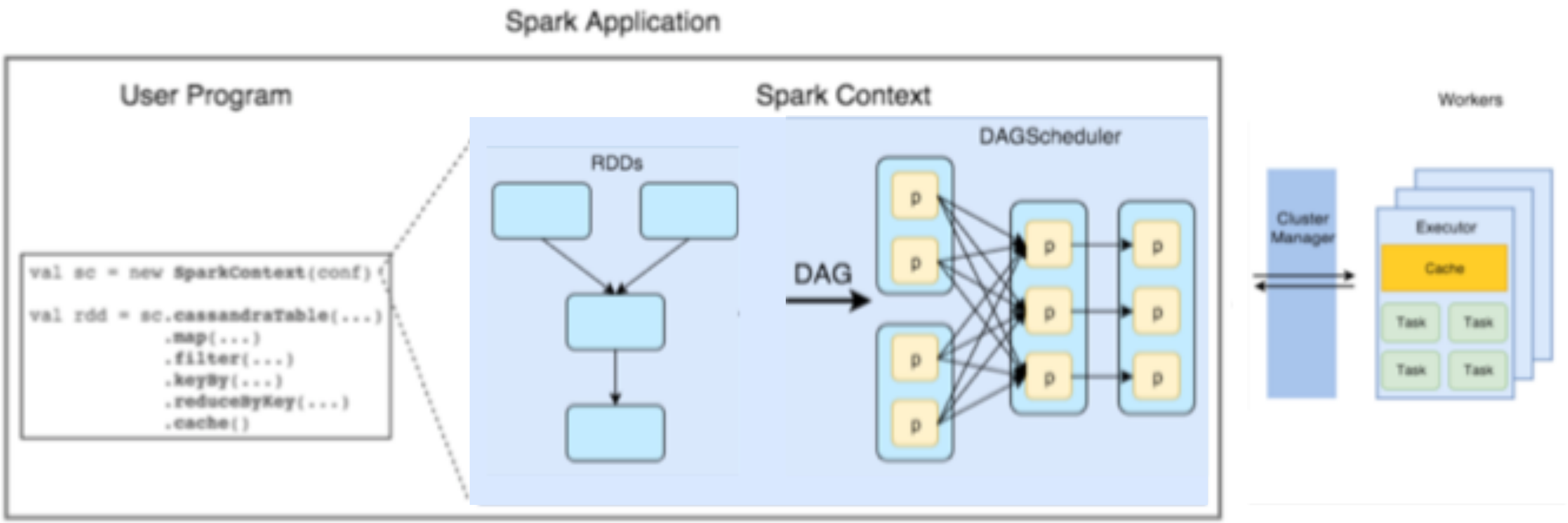
Flint

- Flint: prototype execution engine for serverless PySpark
  - PySpark with serverless backend by simply specifying a config file
  - No costs for idle capacity
  - Simplicity
  - Use cases: ad hoc analytics and exploratory data analysis

Flint architecture

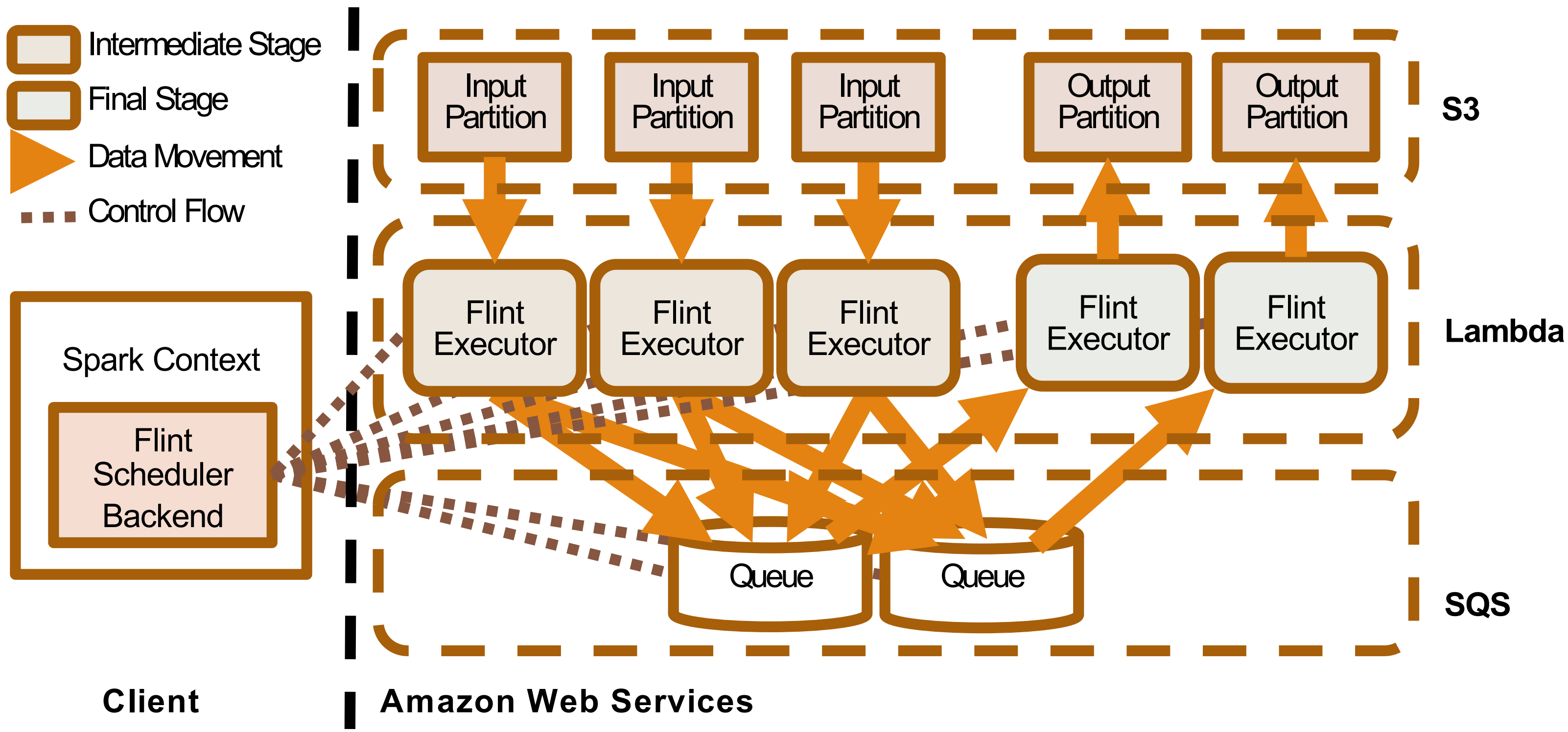
- Spark tasks are executed in AWS Lambda
- Intermediate data are held in Amazon’s Simple Queue Service (SQS)
- Reuses as many existing Spark components as possible
  - Query planning and optimization
  - Many different types of RDD transformations

Workflow



Spark Application
User Program
Spark Context
Workers
DAGScheduler
RDDs
DAG
Cluster Manager
Executor
Cache
Task
Task
Task
Task
val sc = new SparkContext(conf)
val rdd = sc.cassandraTable(...)
.map(...)
.filter(...)
.keyBy(...)
.reduceByKey(...)
.cache()

Flint architecture



Flint architecture

- The Flint scheduler coordinates Flint executors to execute a particular physical plan
  - Function registration
  - Queue initialization
  - Serialization
  - Invocation using thread pool
  - Process the response from an executor

Flint executor

- Flint executor is a python process running inside an Amazon Lambda function
- Each serverless compute function invocation processes a single task
  - Simplifies the communication requirement between an executor and a driver
  - Less affected by the limitation of execution time

Remote storage for shuffling

- No permanent storage
- Small ephemeral disk space (~512 MB)
- Execution time limitation

  => Cannot guarantee the Flint executors from the previous stage are still alive to pass data
- Communication between Lambda functions

- Amazon's Simple Queue Service (SQS)
  - highly-scalable
  - reliable

Experimental Evaluation

- A Spark cluster running the Databricks Unified Analytics Platform (Standard)
- 11 m4.2xlarge instances (one driver and ten workers) - 80 vCores
- 80 max concurrent invocations (~ 80 vCores)

Experimental Evaluation

- NYC taxi dataset (215 GB)
- Pick-up and drop-off dates/time, trip distance, payment type, tip amount
- Queries inspired by an exploratory data analysis task described in a popular blog post by Todd Schneider

Experimental Evaluation

- Q0: Line count
- Q1: Taxi drop-offs at the Goldman Sachs headquarters (hourly aggregation)

```
arr = src.map(lambda x: x.split(',')) \
.filter(lambda x: inside(x, goldman)) \
.map(lambda x: (get_hour(x), 1)) \
.reduceByKey(add, 30) \
.collect()
```

Experimental Evaluation

- Q2: Similar to Q1, but for Citigroup headquarters
- Q3: Goldman Sachs taxi drop-offs with tips greater than $10
- Q4: Cash vs. credit card payments
- Q5: Yellow taxi vs. green taxi , monthly aggregation
- Q6: Effect of precipitation on taxi trips

Experimental Evaluation

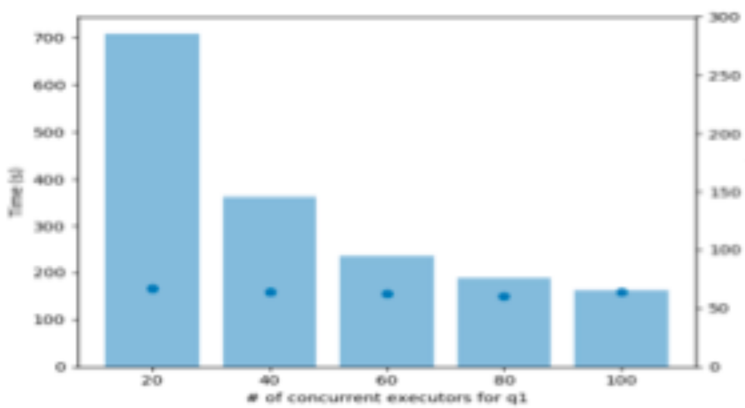
	Query Latency (s)			Estimated Cost (USD)		
	Flint	**PySpark**	**Spark**	**Flint**	**PySpark**	**Spark**
0	101 [93 - 109]	211	188	0.20	0.41	0.37
1	190 [186 - 197]	316	189	0.59	0.61	0.37
2	203 [201 - 205]	314	187	0.68	0.61	0.36
3	165 [161 - 169]	312	188	0.48	0.61	0.36
4	132 [122 - 142]	225	189	0.33	0.44	0.37
5	159 [142 - 177]	312	189	0.45	0.60	0.37
6	277 [272 - 281]	337	191	0.56	0.66	0.37

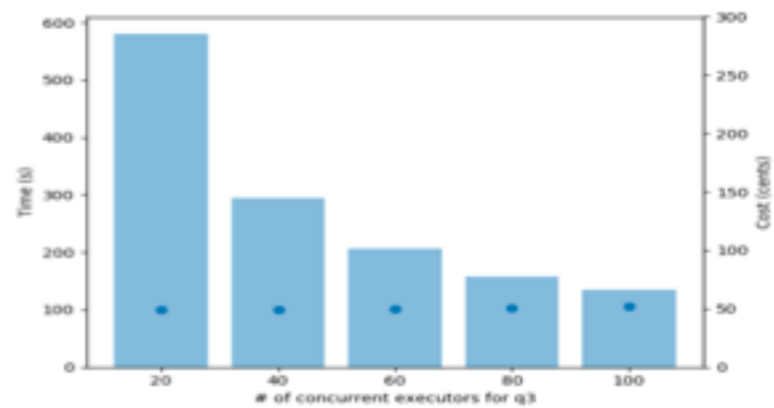
TABLE I
QUERY LATENCY AND COST COMPARISONS.

Experimental Evaluation

- Q1: Taxi drop-offs at the Goldman Sachs headquarters (hourly aggregation)
- Q3: Goldman Sachs taxi drop-offs with tips greater than $10

tradeoff of concurrency between the latency and the cost



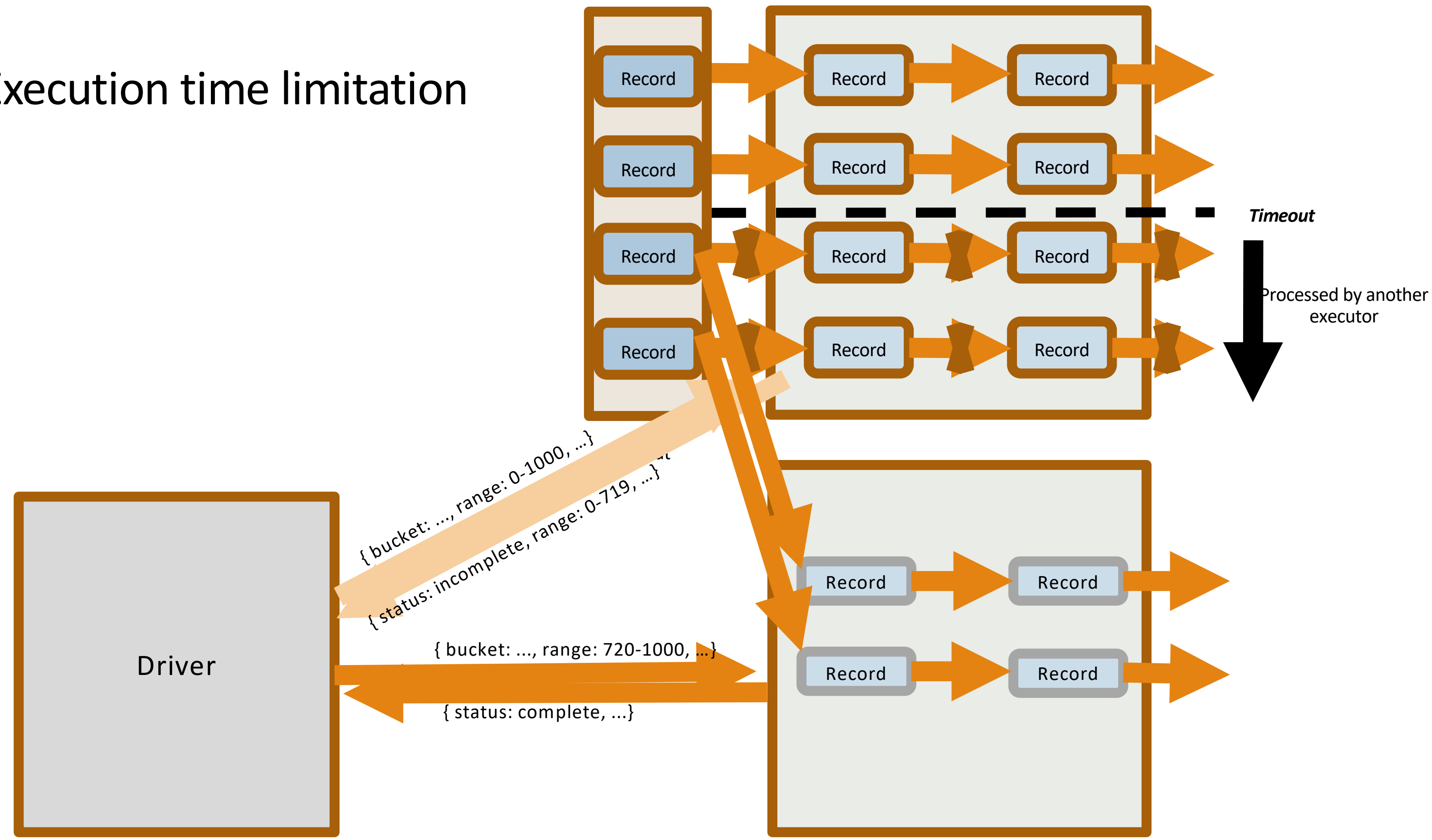


Lambda Limitations

- Most serverless platforms currently have several limitations
  - Memory size (e.g. 3008 MB for AWS)
  - Execution time limitation (5 ~ 9 minutes)
  - Cold start problem

Lambda Limitations

Execution time limitation



Lambda Limitations

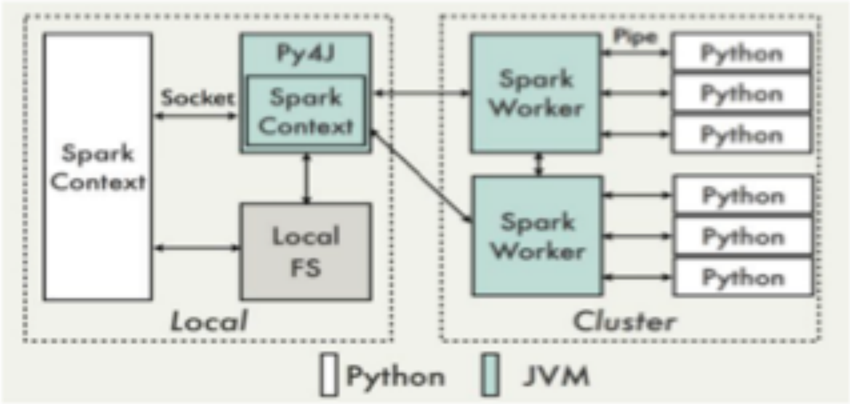


Figure 3.4: PySpark internals on standard Spark cluster [13]. Workers rely on Java/Scala Spark running on JVM

Lambda Limitations

Other constraints

- Memory (3008 MB)
- Request size: 6 MB
  - Metadata
- Response size: 6 MB
  - Collect

Related Work

- Iris:
  - The origin of Flint (A course project, UWaterloo, Fall 2016)
  - Distributed computation framework supporting a subset of Spark API
  - In-browser data analytics backed by serverless backend
- Amazon Athena
  - Per-query pricing with zero idle costs
  - Only supports SQL
  - Presto distributed SQL engine
- Databricks Serverless
  - Automatically managed pools of cloud resources
    - auto-configured & auto-scaled

Related Work

- PyWren
  - Framework built from scratch on top of serverless compute functions and persistent storage
- Qubole Spark on Serverless
  - Ported the existing Spark executor infrastructure onto AWS Lambda, whereas Flint is a from-scratch implementation
    - Communication model
    - AWS Lambda limitations

Future Work

- Intensive shuffling tasks
- Robustness
- Higher level libraries (e.g. MLlib)