Decomposing SMACK Stack

Spark & Mesos Internals

Anton Kirillov
intro by Sebastian Stoll

Apache Spark Meetup
Oooyala, March 2016

Who is this guy?

@antonkirillov

- Staff Engineer in Data Team @ Ooyala
- Scala programmer
- Focused on distributed systems
- Building data platforms with SMACK/Hadoop
- Ph.D. in Computer Science

- blog: datastrophic.io
- github: github.com/datastrophic

Roadmap

- Intro to Ooyala stack
- SMACK stack
  - Overview
  - Architecture design options
- Spark
  - Core concepts & execution workflow
  - Architecture
- Mesos
  - Cluster resource management
  - Architecture and scheduling
  - Frameworks
  - Spark on Mesos


OOYALA
REVOLUTIONIZING
DIGITAL TV

What Ooyala does

Program Management

Signal/Asset Acquisition

Encoding and Packaging

App Infrastructure and Clients

Authentication

Playback Experience

Delivery Infrastructure

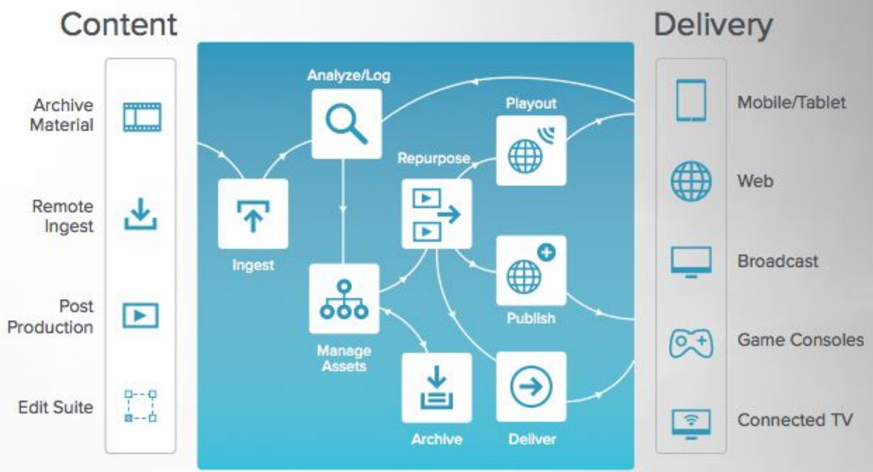
Content Recommendations

Monetization

Analytics and Insights

Online Video Platform

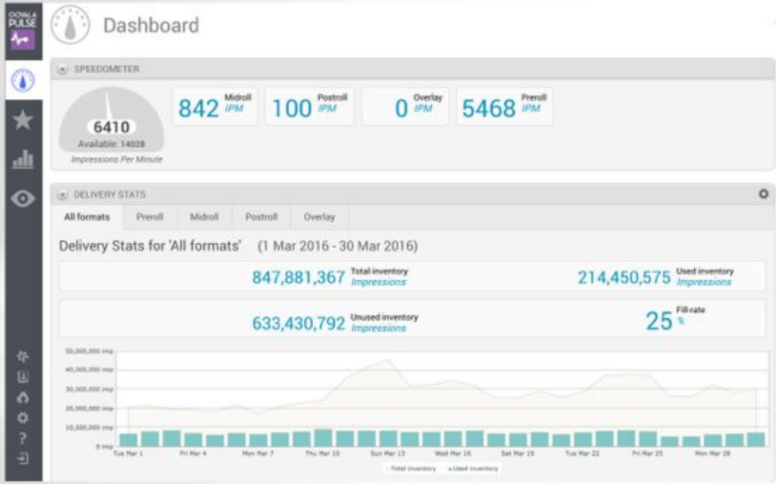
MIO



Ooyala Media Logistics

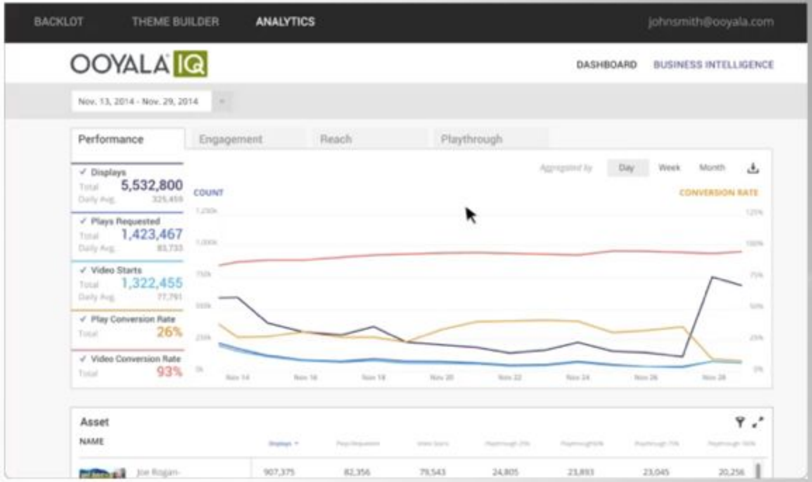
What Ooyala does

OOYALA® PULSE



Ad Tech

OOYALA® IQ



Analytics

Ooyala does that

Spark

- Ooyala IQ
- Data operations at scale
- Stream analytics

MESOS

- Microservice infrastructure
- Spark cluster resource manager

akka

- Load testing
- General tooling

cassandra

- Ooyala IQ
- Ooyala Pulse Insight
- Delivery metrics
- Forecasting

kafka

- Ooyala Now
- Data Flow

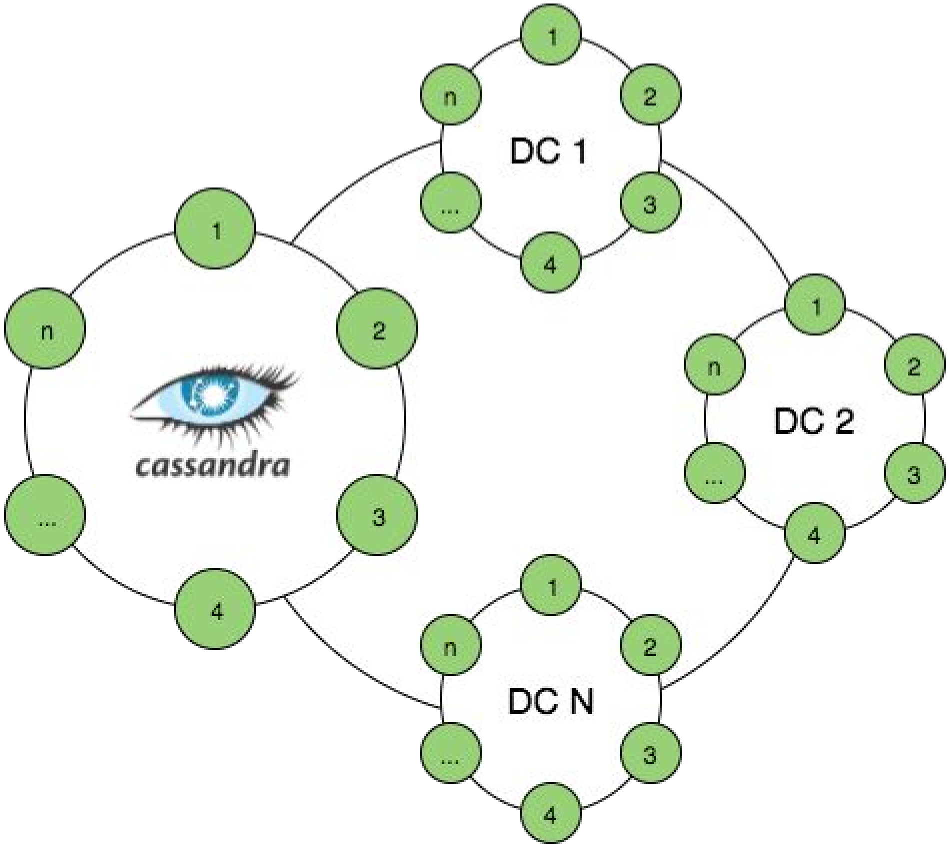
SMACK Stack Overview

components and architecture designs

SMACK Stack

- **Spark** - a generalized framework for distributed data processing supporting in-memory data caching and reuse across computations
- **Mesos** - cluster resource management system that provides efficient resource isolation and sharing across distributed applications
- **Akka** - a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM
- **Cassandra** - distributed, highly available database designed to handle large amounts of data across multiple datacenters
- **Kafka** - a high-throughput, low-latency distributed messaging system designed for handling real-time data feeds

Storage Layer: Cassandra

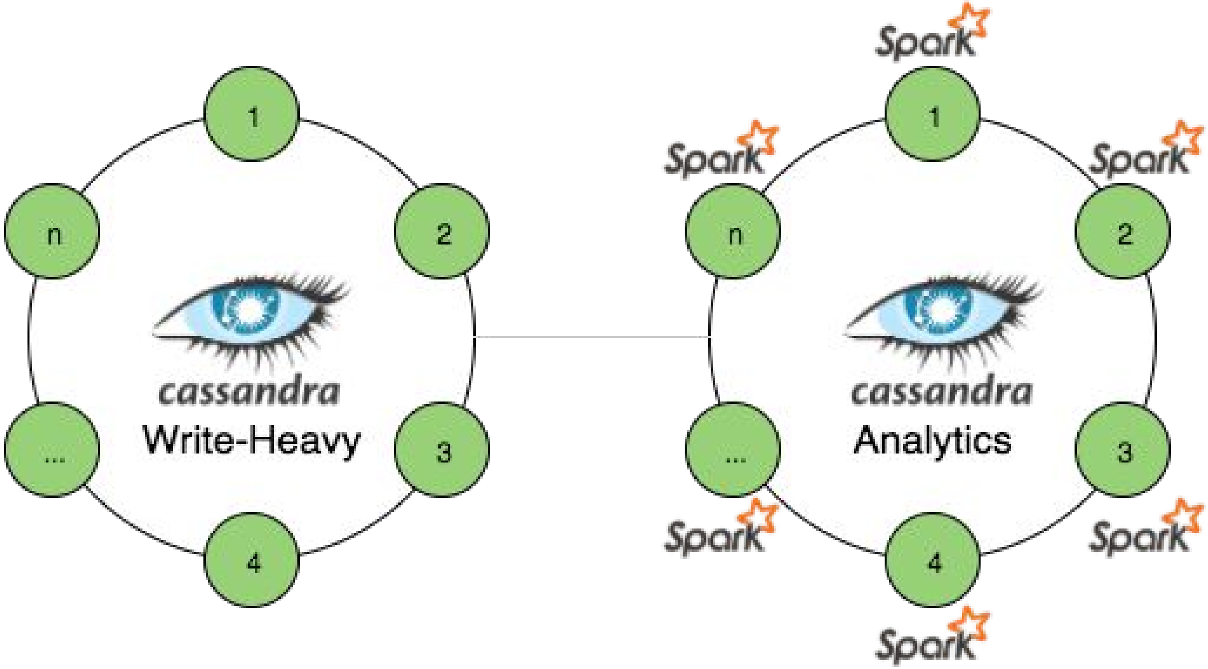


- Pros:
  - optimized for heavy write loads
  - configurable CA (CAP)
  - linearly scalable
  - XDCR support
  - easy cluster resizing and inter-DC data migration
- Cons:
  - data model (distributed nested sorted map)
  - designed for fast serving but not batch processing
  - not well-suited for ad-hoc queries against historical raw data

Fixing NoSQL limitations with Spark

```
//joining raw events with rolled-up and grouping by type
sqlContext.sql {"""
    SELECT
    events.campaignId,
    events.eventType,
    events.value + campaigns.total as total_events
    FROM events
    JOIN campaigns
    ON events.campaignId = campaigns.id AND events.eventType = campaigns.eventType
    """.stripMargin
}.registerTempTable("joined")

sqlContext.sql {"""
    SELECT campaignId, eventType, sum(total_events) as total
    FROM joined
    GROUP BY campaignId, eventType
    """.stripMargin
}.saveAsCassandraTable(”keyspace”, ”totals”)
```

Architecture of Spark/Cassandra Clusters

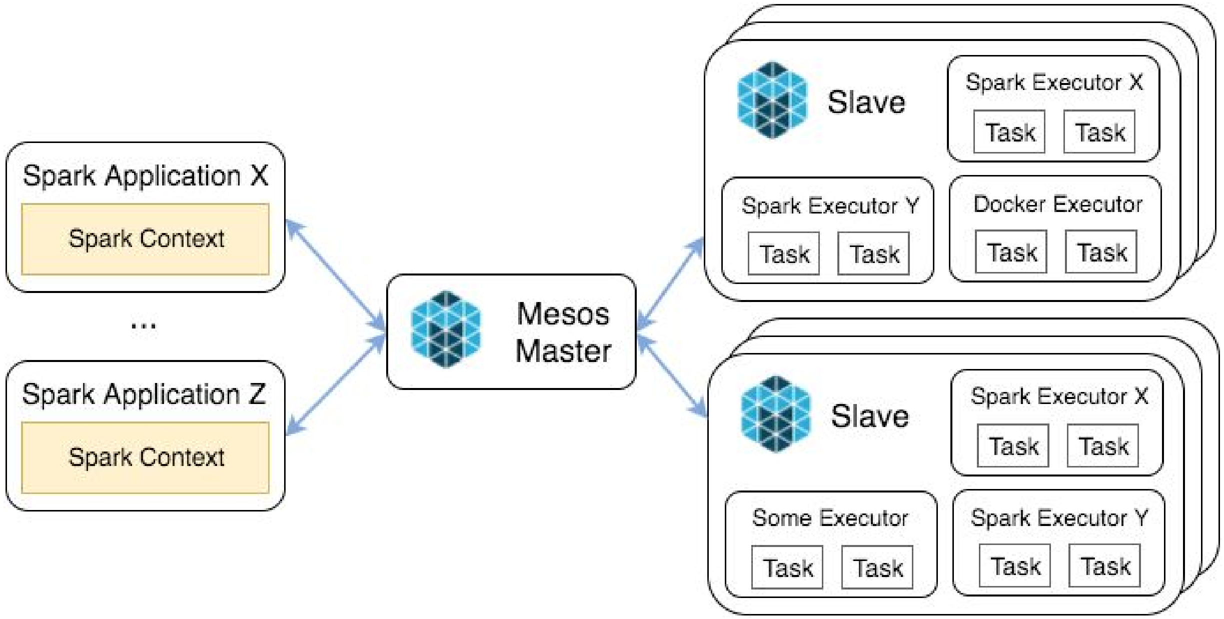


Separate Write & Analytics:

- clusters can be scaled independently
- data is replicated by Cassandra asynchronously
- Analytics has different Read/Write load patterns
- Analytics contains additional data and processing results
- Spark resource impact limited to only one DC

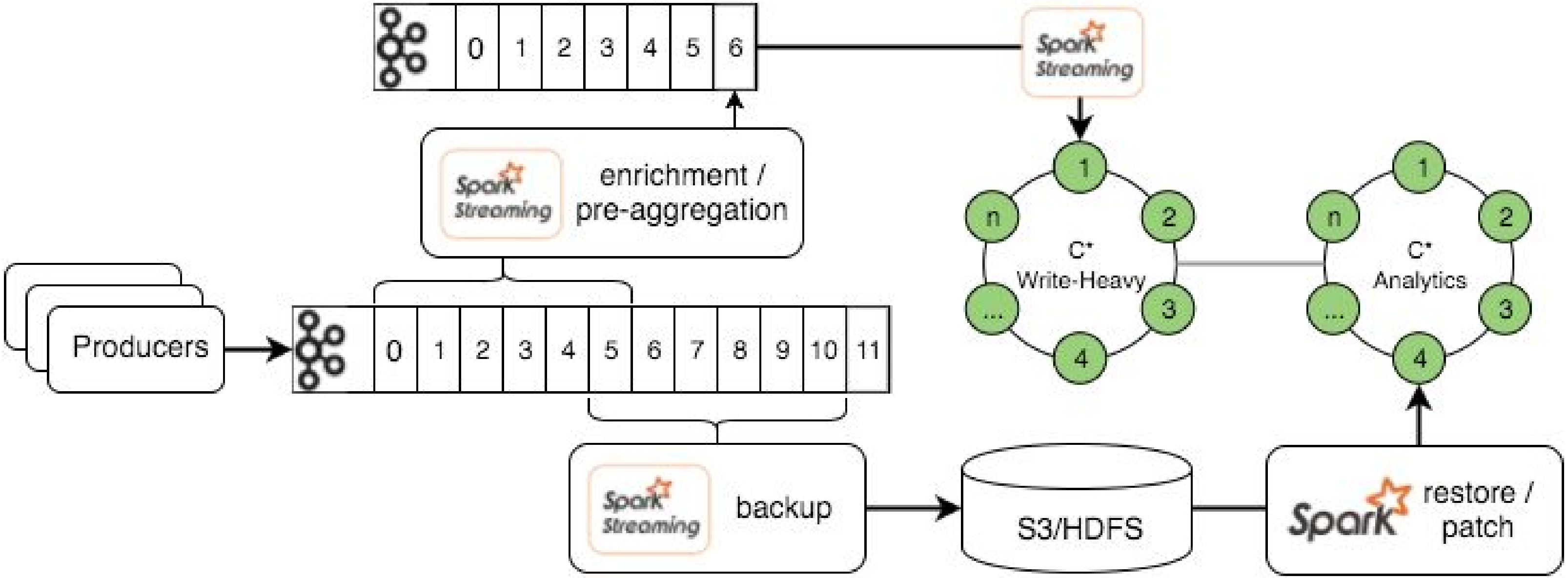
To fully facilitate Spark-C* connector data locality awareness,
Spark workers should be collocated with Cassandra nodes (**gotcha**: CL=ONE)

Mesos as Spark cluster manager



- fine-grained resource sharing between Spark and other applications
- scalable partitioning between multiple instances of Spark
- unified platform for running various applications (frameworks)
- fault-tolerant and scalable

Stream Processing with Kafka and Spark

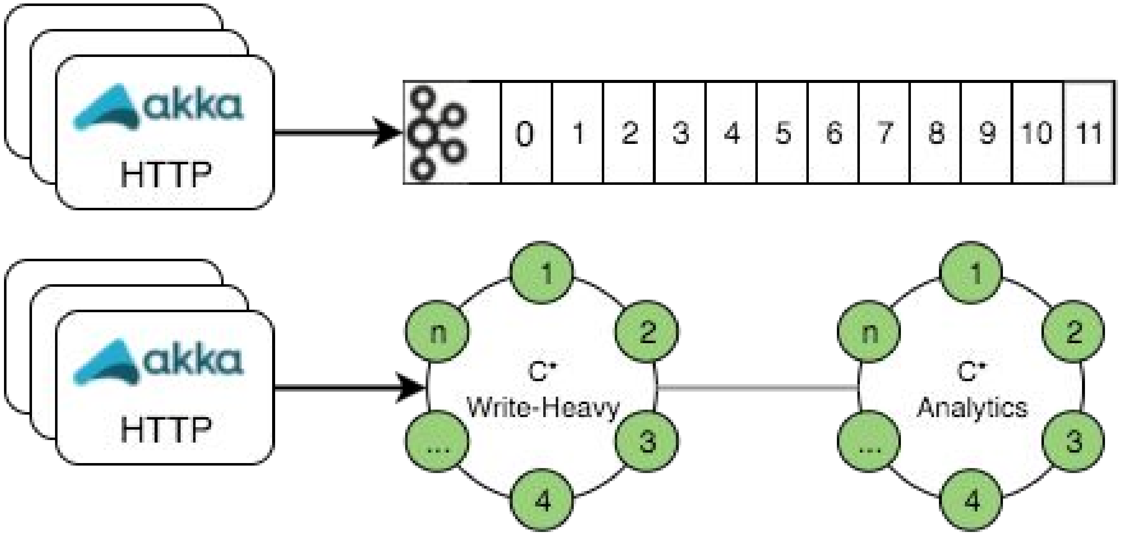


- be prepared for failures and broken data
- backup and patching strategies should be designed upfront
- patch/restore if time interval could be done by replay if store is idempotent

Spark Streaming with Kafka

```
val streamingContext = new StreamingContext(sc.getConf, Seconds(10))

val eventStream = KafkaUtils.createStream(
  ssc = streamingContext,
  zkQuorum = "zoo01,zoo02,zoo03",
  groupId = "spark_consumer",
  topics = Map("raw_events" -> 3)
)

eventStream.map(_.toEvent)
                    .saveToCassandra(keyspace, table)

streamingContext.start()
streamingContext.awaitTermination()
```

Data Ingestion with Akka



- actor model implementation for JVM
- message-based and asynchronous
- easily scalable from one process to cluster of machines
- actor hierarchies with parental supervision
- easily packages in Docker to be run on Mesos

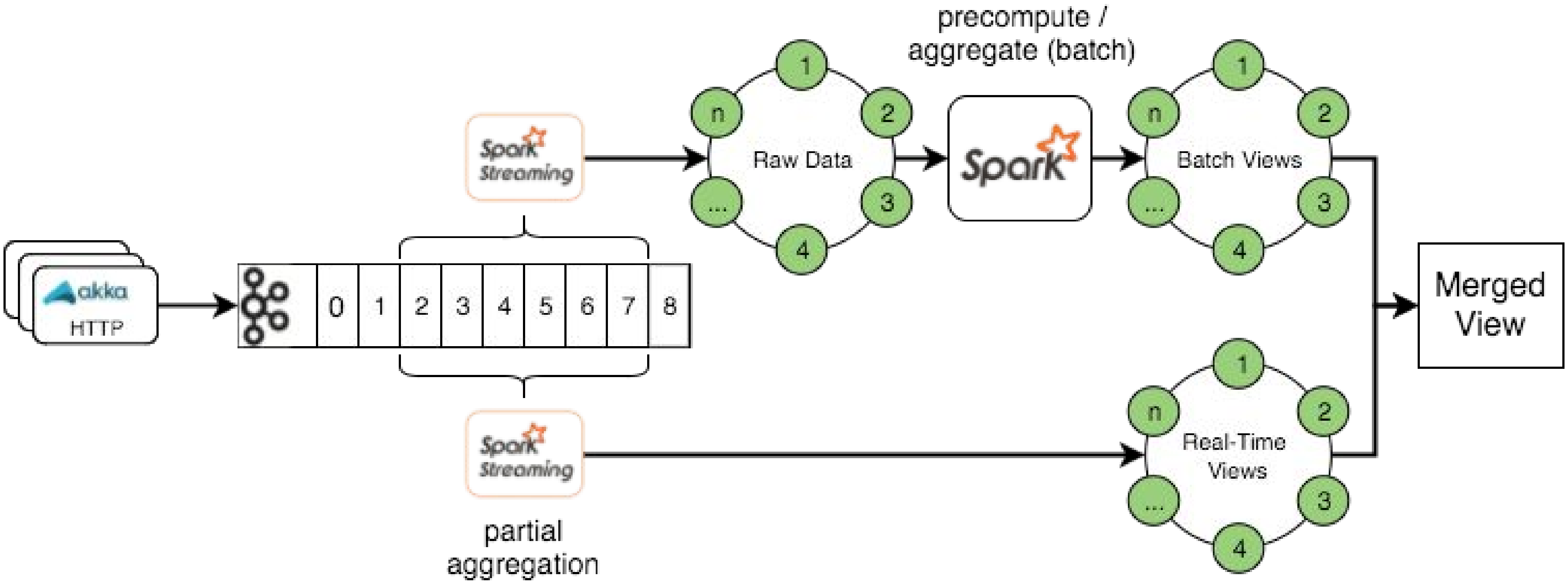
Akka Http microservice

```scala
val config = new ProducerConfig(KafkaConfig())
lazy val producer = new KafkaProducer[A, A](config)

val routes: Route = {
  post{
     decodeRequest{
        entity(as[String]){ str =>
           JsonParser.parse(str).validate[Event] match {
              case s: JsSuccess[String] =>
                 producer.send(new KeyedMessage(topic, str))
                 system.actorOf(Props[CassandraWriter]) ! s.get
              case e: JsError => BadRequest -> JsError.toFlatJson(e).toString()
           }
        }
     }
  }
}

object AkkaHttpMicroservice extends App with Service {
  Http().bindAndHandle(routes, config.getString("http.interface"), config.getInt("http.port"))
}
```

Writing to Cassandra with Akka

```scala
class CassandraWriterActor extends Actor with ActorLogging {

  //for demo purposes, session initialized here
  val session = Cluster.builder()
              .addContactPoint("cassandra.host")
              .build()
              .connect()

  override def receive: Receive = {
    case event: Event =>
      val statement = new SimpleStatement(event.createQuery)
                  .setConsistencyLevel(ConsistencyLevel.QUORUM)

      Try(session.execute(statement)) match {
        case Failure(ex) => //error handling code
        case Success => sender ! WriteSuccessfull
      }
  }
}
```

Lambda Architecture with SMACK



- when design meets reality it’s hard to implement canonical architecture
- depending on the use case it’s easy to implement Kappa architecture as well

SMACK stack:

- concise toolbox for wide variety of data processing scenarios
- battle-tested and widely used software with large communities
- easy scalability and replication of data while preserving low latencies
- unified cluster management for heterogeneous loads
- single platform for any kind of applications
- implementation platform for different architecture designs
- really short time-to-market (e.g. for MVP verification)

Apache Spark in Depth

core concepts, architecture & internals

Meet Spark

- Generalized framework for distributed data processing (batch, graph, ML)
- Scala collections functional API for manipulating data at scale
- In-memory data caching and reuse across computations
- Applies set of coarse-grained transformations over partitioned data
- Failure recovery relies on lineage to recompute failed tasks
- Supports majority of input formats and integrates with Mesos / YARN

Spark makes data engineers happy

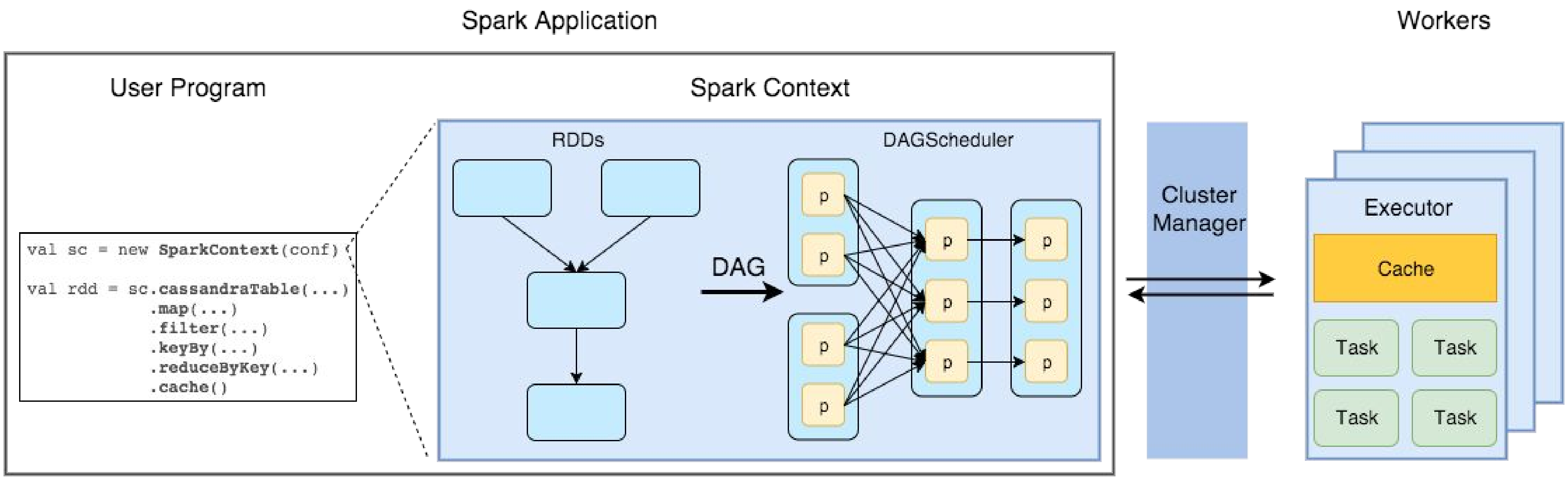
Backup/restore of Cassandra tables in Parquet

```
def backup(config: Config) {
  sc.cassandraTable(config.keyspace, config.table).map(_.toEvent).toDF()
  .write.parquet(config.path)
}

def restore(config: Config) {
  sqlContext.read.parquet(config.path)
  .map(_.toEvent).saveToCassandra(config.keyspace, config.table)
}
```

Query different data sources to identify discrepancies

```
sqlContext.sql {
  """
    SELECT count()
    FROM cassandra_event_rollups
    JOIN mongo_event_rollups
    ON cassandra_event_rollups.uuid = cassandra_event_rollups.uuid
    WHERE cassandra_event_rollups.value != cassandra_event_rollups.value
  """.stripMargin
}
```

Core Concepts



RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure
- Provides API for
  - manipulating the collection of elements (transformations and materialization)
  - persisting intermediate results in memory for later reuse
  - controlling partitioning to optimize data placement
- Can be created through deterministic operation
  - from storage (distributed file system, database, plain file)
  - from another RDD
- Stores information about parent RDDs
  - for execution optimization and operations pipelining
  - to recompute the data in case of failure

RDD: a developer's view

- Distributed immutable data + lazily evaluated operations
  - partitioned data + iterator
  - transformations & actions
- An interface defining 5 main properties

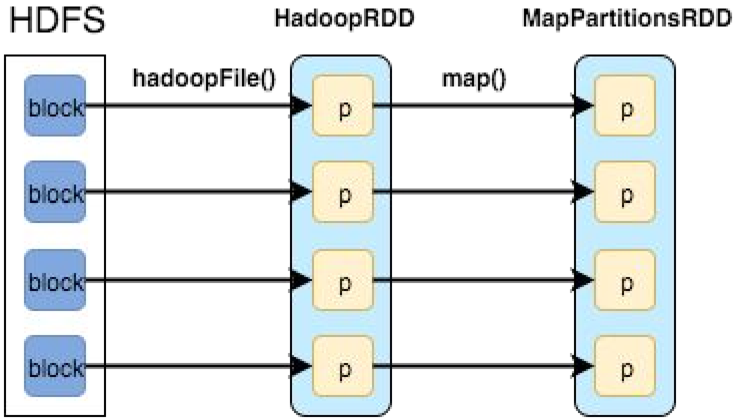
```
a list of partitions (e.g. splits in Hadoop)
def getPartitions: Array[Partition]

a list of dependencies on other RDDs
def getDependencies: Seq[Dependency[_]]

a function for computing each split
def compute(split: Partition, context: TaskContext): Iterator[T]
```

lineage

```
(optional) a list of preferred locations to compute each split on
def getPreferredLocations(split: Partition): Seq[String] = Nil

(optional) a partitioner for key-value RDDs
val partitioner: Option[Partitioner] = None
```

execution optimization

RDDs Example



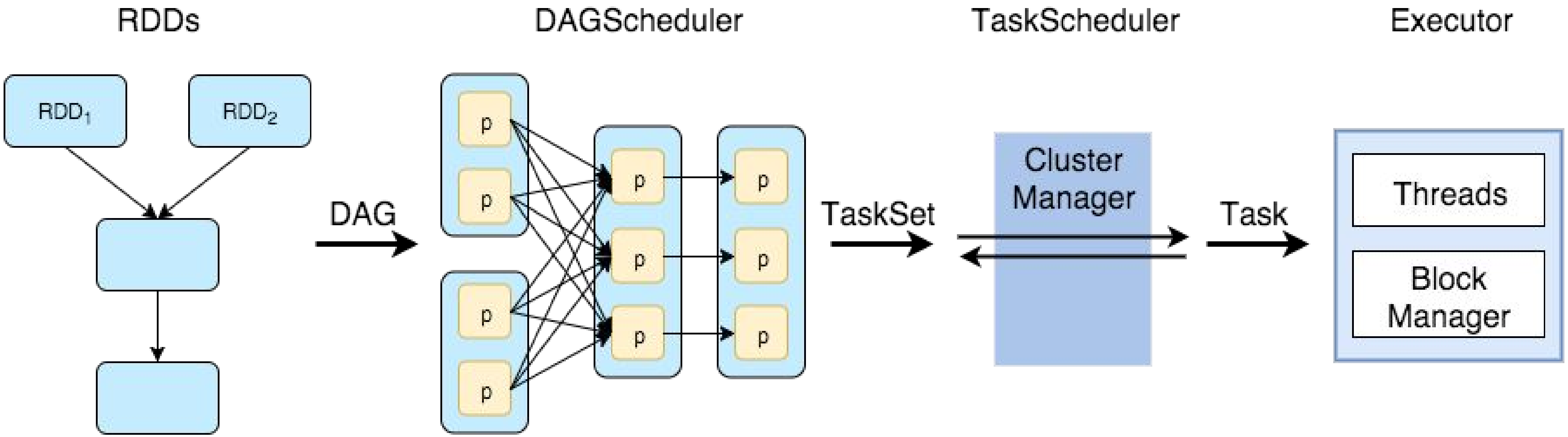
sparkContext.textFile("hdfs://...")

- HadoopRDD
  - getPartitions = HDFS blocks
  - getDependencies = None
  - compute = load block in memory
  - getPrefferedLocations = HDFS block locations
  - partitioner = None
- MapPartitionsRDD
  - getPartitions = same as parent
  - getDependencies = parent RDD
  - compute = compute parent and apply map()
  - getPrefferedLocations = same as parent
  - partitioner = None

RDD Operations

- Transformations
  - apply user function to every element in a partition (or to the whole partition)
  - apply aggregation function to the whole dataset (groupBy, sortBy)
  - introduce dependencies between RDDs to form DAG
  - provide functionality for repartitioning (repartition, partitionBy)
- Actions
  - trigger job execution
  - used to materialize computation results
- Extra: persistence
  - explicitly store RDDs in memory, on disk or off-heap (cache, persist)
  - checkpointing for truncating RDD lineage

Execution workflow



rdd1.**join**(rdd2)
    .**groupBy**(...)
    .**filter**(...)

splits graph into *stages* of tasks

submits each stage as ready

launches tasks via cluster manager

retries failed or struggling tasks

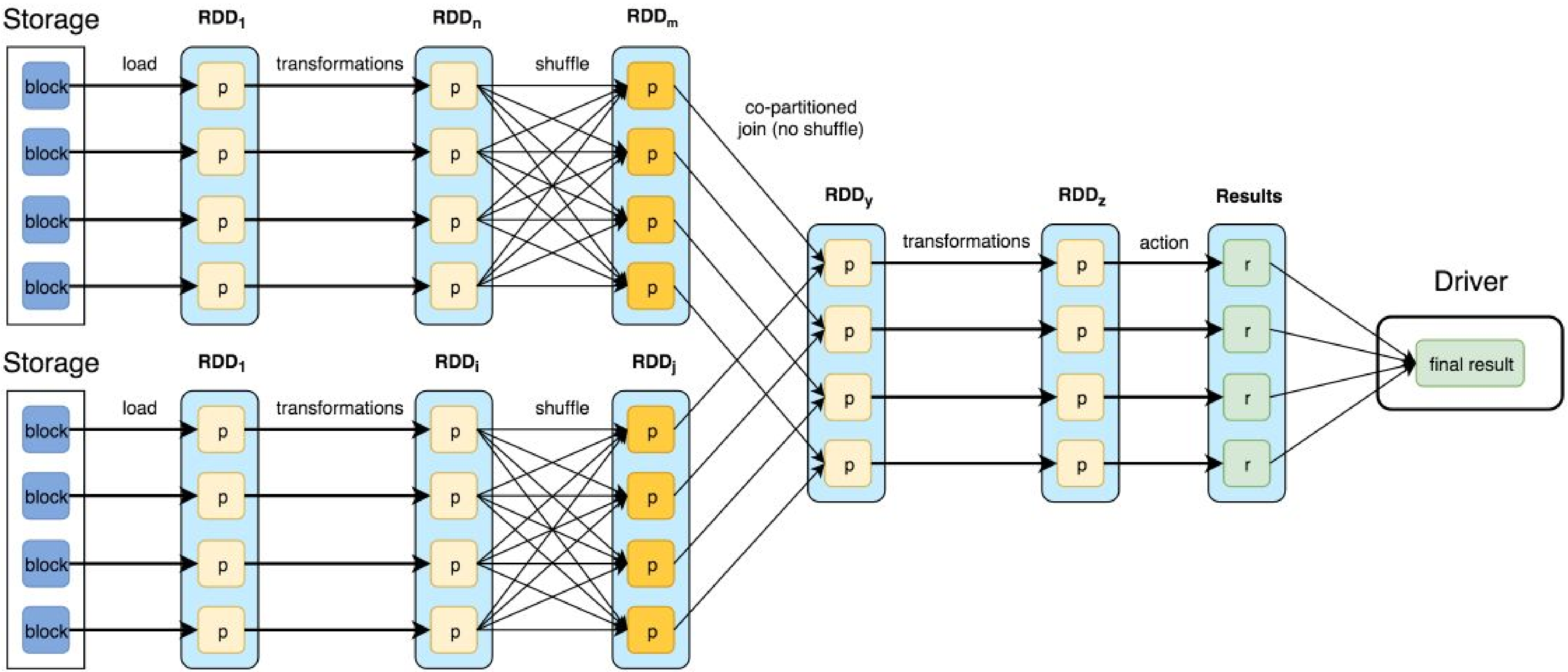
executes tasks

stores and serves blocks

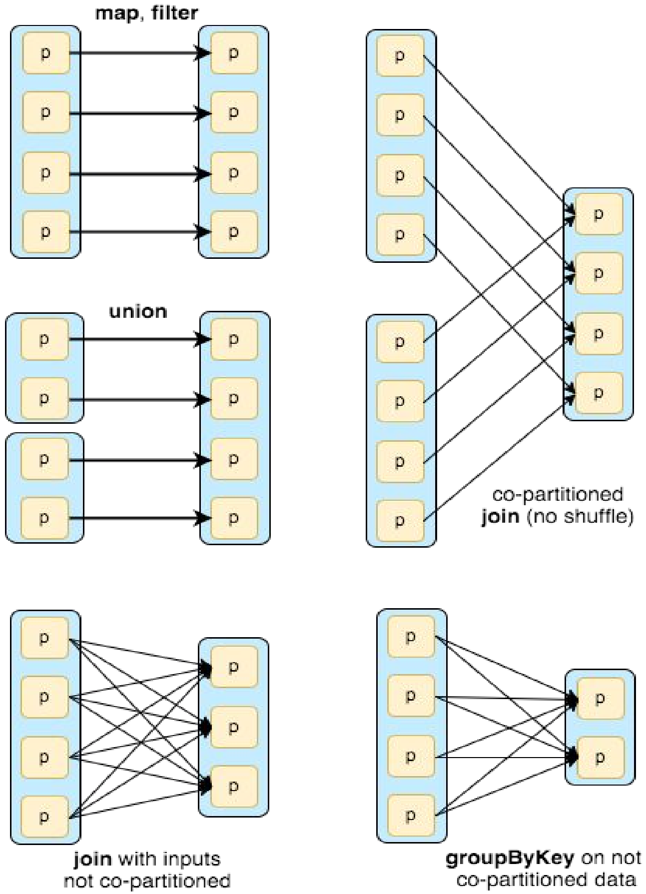
Code sample: joining aggregated and raw data

```scala
//aggregate events after specific date for given campaign
val events =  sc.cassandraTable("demo", "event")
                 .map(_.toEvent)
                 .filter(event => event.campaignId == campaignId && event.time.isAfter(watermark))
                 .keyBy(_.eventType)
                 .reduceByKey(_ + _)
                 .cache()

//aggregate campaigns by type
val campaigns = sc.cassandraTable("demo", "campaign")
                   .map(_.toCampaign)
                   .filter(campaign => campaign.id == campaignId && campaign.time.isBefore(watermark))
                   .keyBy(_.eventType)
                   .reduceByKey(_ + _)
                   .cache()

//joined rollups and raw events
val joinedTotals = campaigns.join(events)
                          .map { case (key, (campaign, event)) => CampaignTotals(campaign, event) }
                          .collect()
//count totals separately
val eventTotals = events.map{ case (t, e) => s"$t -> ${e.value}" }.collect()
val campaignTotals = campaigns.map{ case (t, e) => s"$t -> ${e.value}" }.collect()
```

DAG

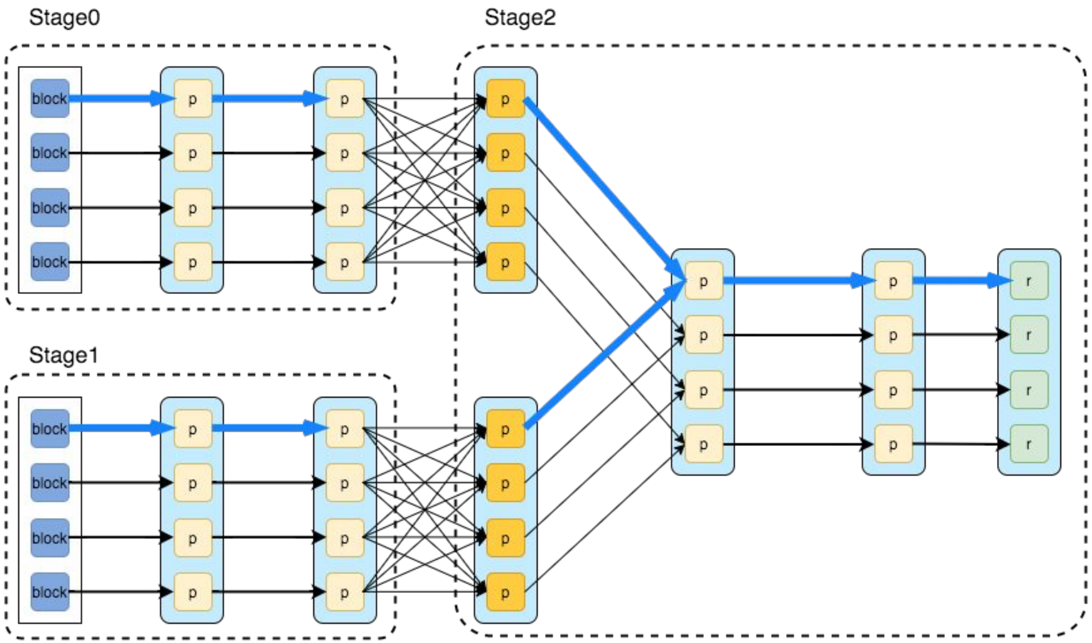

Storage
block
block
block
block
load
RDD1
p
transformations
RDDn
shuffle
RDDm
co-partitioned
join (no shuffle)
Storage
load
RDD1
transformations
RDDi
shuffle
RDDj
RDDy
transformations
RDDz
action
Results
r
Driver
final result

Dependency types



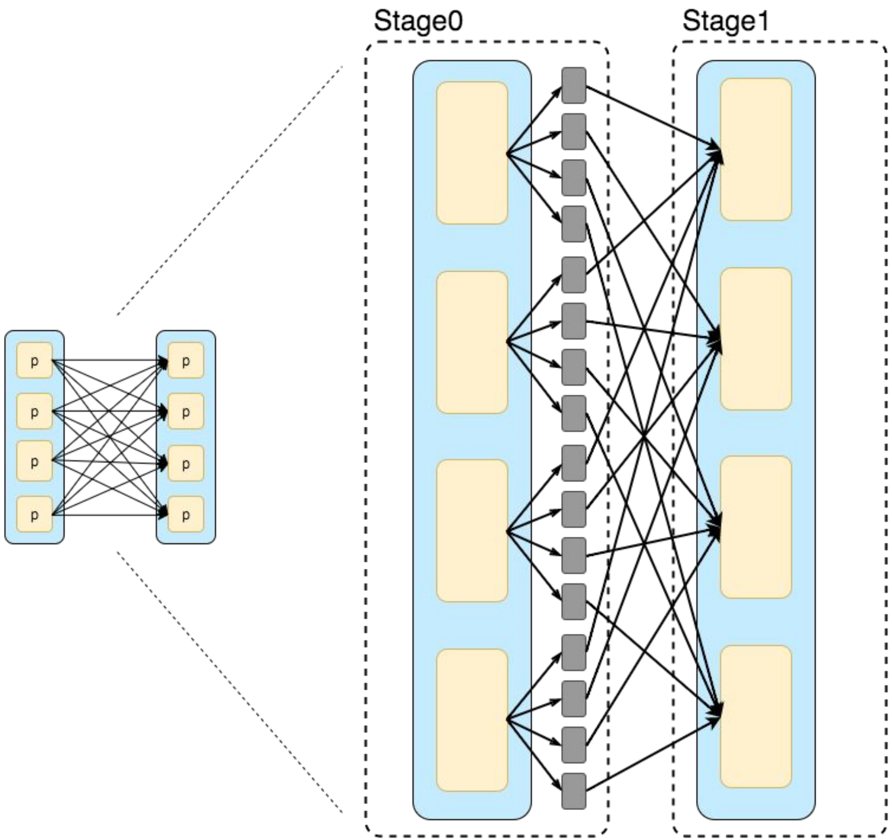
- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed
- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed

Stages and Tasks



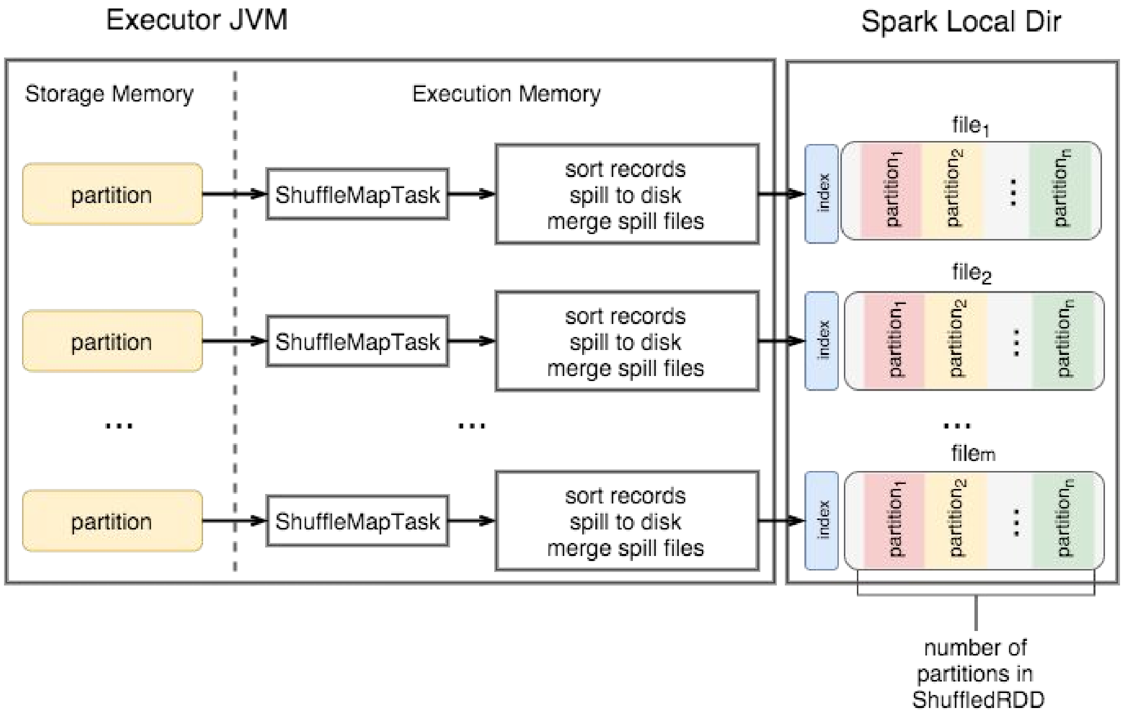
- Stages breakdown strategy
  - check backwards from final RDD
  - add each “narrow” dependency to the current stage
  - create new stage when there’s a shuffle dependency
- Tasks
  - *ShuffleMapTask* partitions its input for shuffle
  - *ResultTask* sends its output to the driver

Shuffle



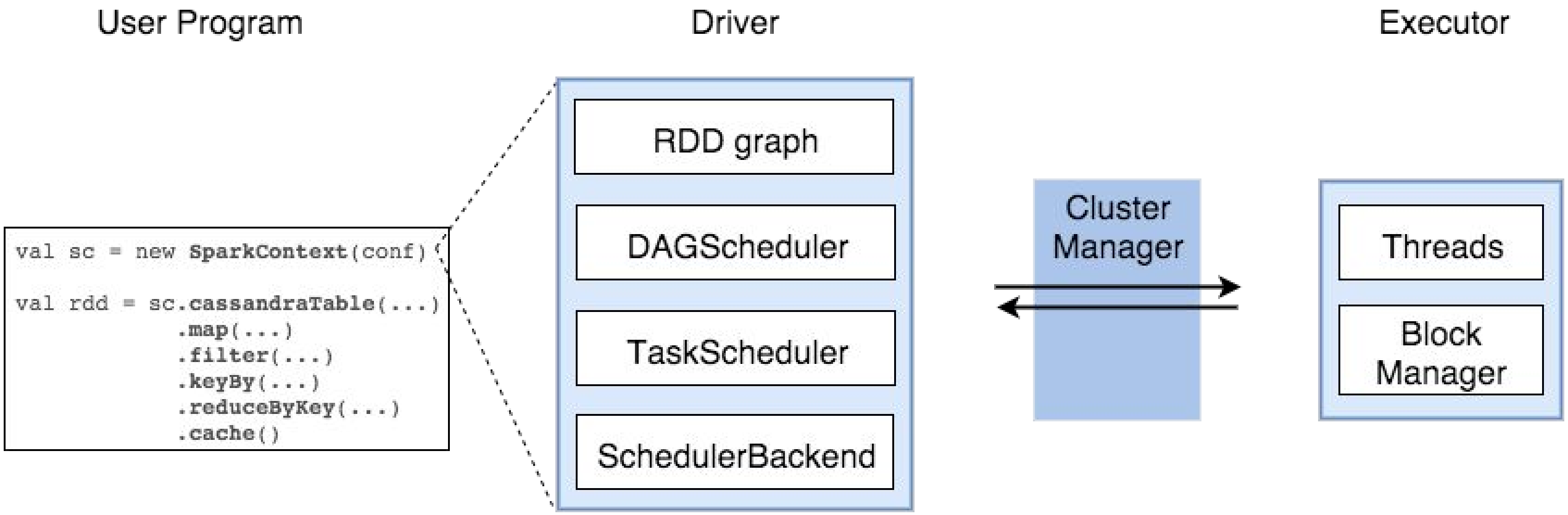
- Shuffle Write
  - redistributes data among partitions and writes files to disk
  - each *shuffle* task creates one file with regions assigned to reducer
  - *sort shuffle* uses in-memory sorting with spillover to disk to get final result
- Shuffle Read
  - fetches the files and applies *reduce()* logic
  - if data ordering is needed then it is sorted on “reducer” side for any type of shuffle

Sort Shuffle

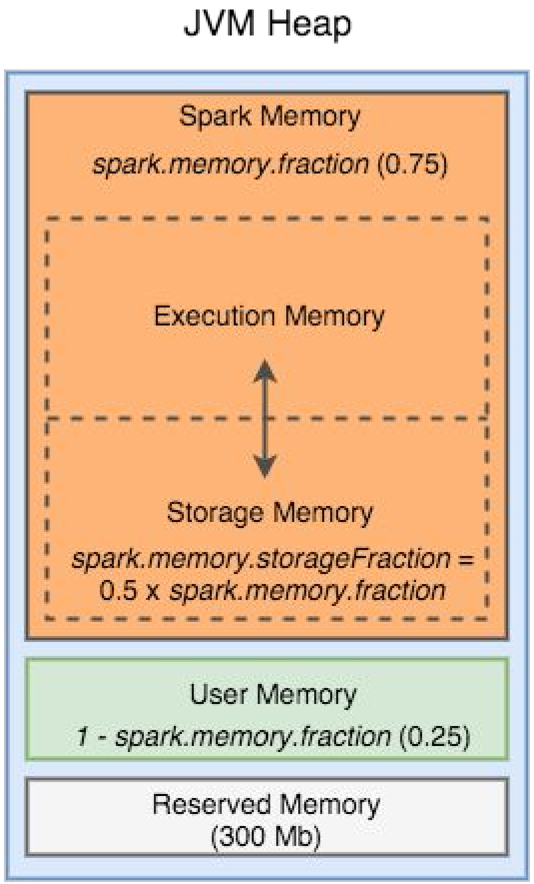


- Incoming records accumulated and sorted in memory according their target partition ids
- Sorted records are written to file or multiple files if spilled and then merged
- *index* file stores offsets of the data blocks in the data file
- Sorting without deserialization is possible under certain conditions (SPARK-7081)

Spark Components



Memory Management in Spark 1.6



- Execution Memory
  - storage for data needed during tasks execution
  - shuffle-related data
- Storage Memory
  - storage of cached RDDs and broadcast variables
  - possible to borrow from execution memory (spill otherwise)
  - safeguard value is 0.5 of Spark Memory when cached blocks are immune to eviction
- User Memory
  - user data structures and internal metadata in Spark
  - safeguarding against OOM
- Reserved memory
  - memory needed for running executor itself and not strictly related to Spark

Execution Modes

- spark-shell --master [local | spark | yarn-client | mesos]
  - launches REPL connected to specified cluster manager
  - always runs in client mode
- spark-submit --master [local | spark:// | mesos:// | yarn] spark-job.jar
  - launches assembly jar on the cluster

- Masters
  - ***local[k]*** - run Spark locally with K worker threads
  - ***spark*** - launches driver app on Spark Standalone installation
  - ***mesos*** - driver will spawn executors on Mesos cluster (***deploy-mode: client | cluster***)
  - **yarn** - same idea as with Mesos (***deploy-mode: client | cluster***)

- Deploy Modes
  - ***client*** - driver executed as a separate process on the machine where it has been launched and spawns executors
  - ***cluster*** - driver launched as a container using underlying cluster manager

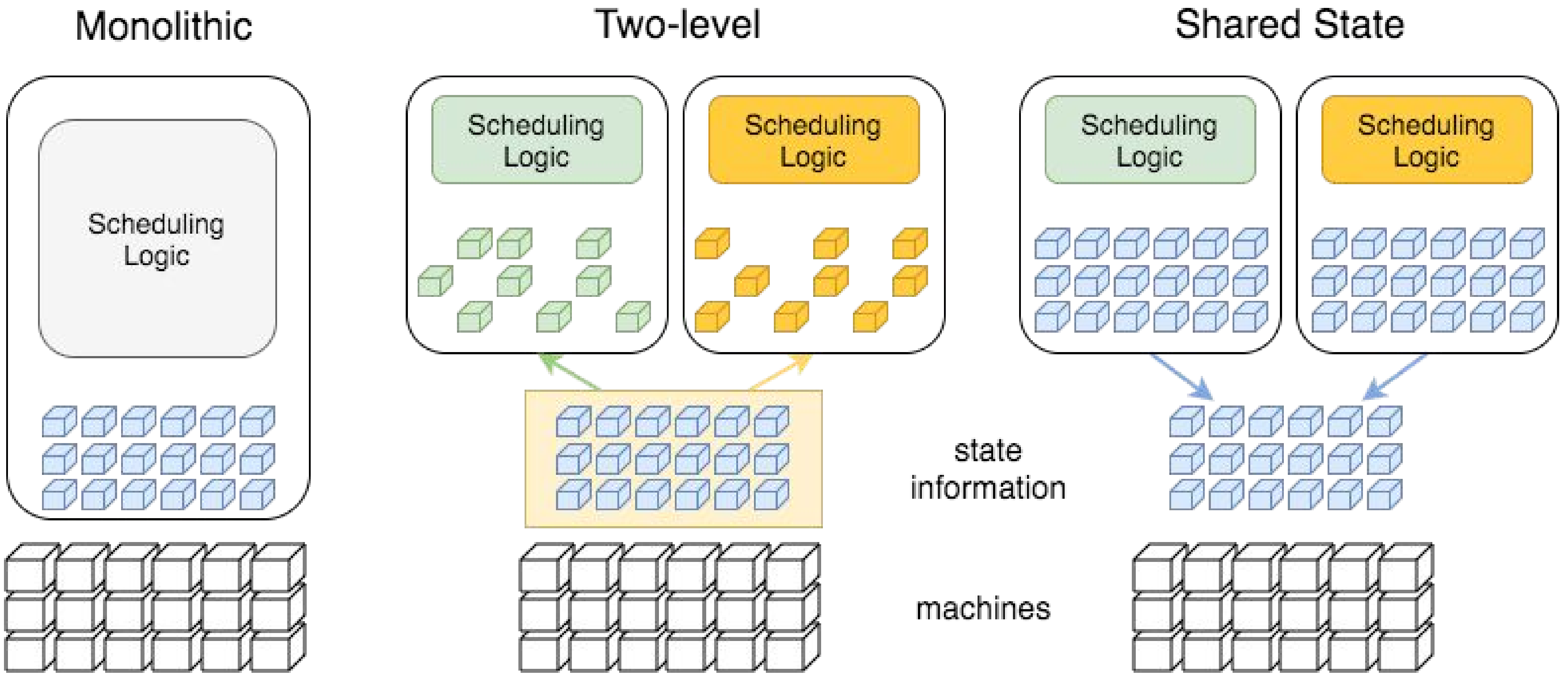
Apache Mesos

architecture, scheduling, frameworks & Spark

Cluster Resource Managers: Requirements

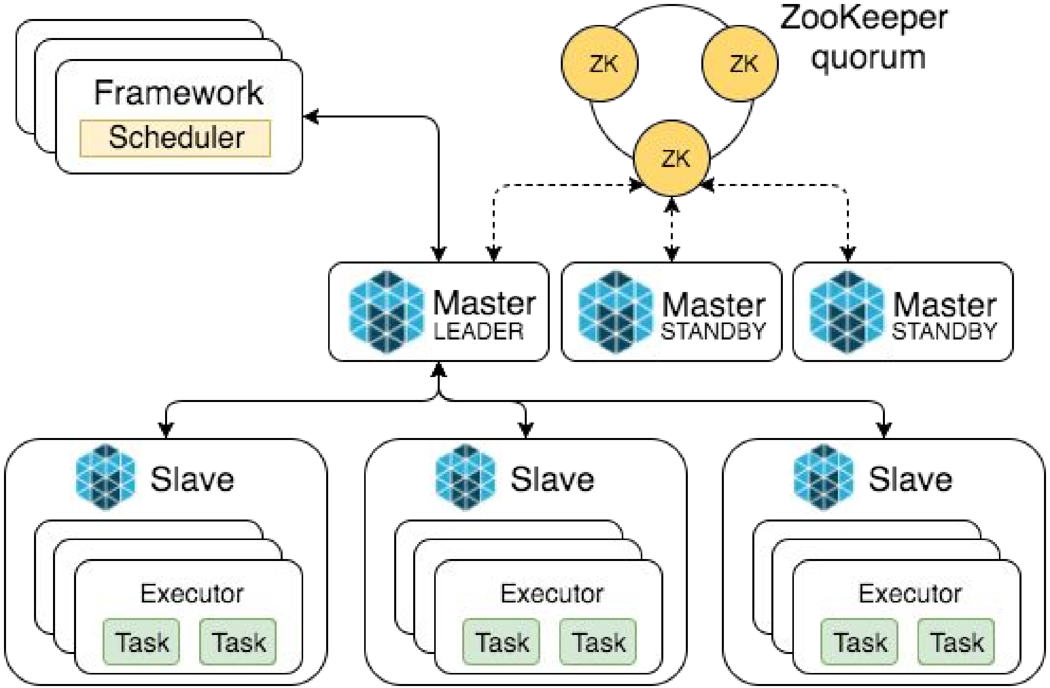
- Efficiency
  - efficient sharing of resources across applications
  - utilization of cluster resources in the most optimal manner
- Flexibility
  - support of wide array of current and future frameworks
  - dealing with hardware heterogeneity
  - support of resource requests of different types
- Scalability
  - scaling to clusters of dozens of thousands of nodes
  - scheduling system’s response times must remain acceptable while increasing number of machines and applications
- Robustness
  - fault-tolerant guarantees for the system and applications
  - high availability of central scheduler component

Cluster Manager Architectures



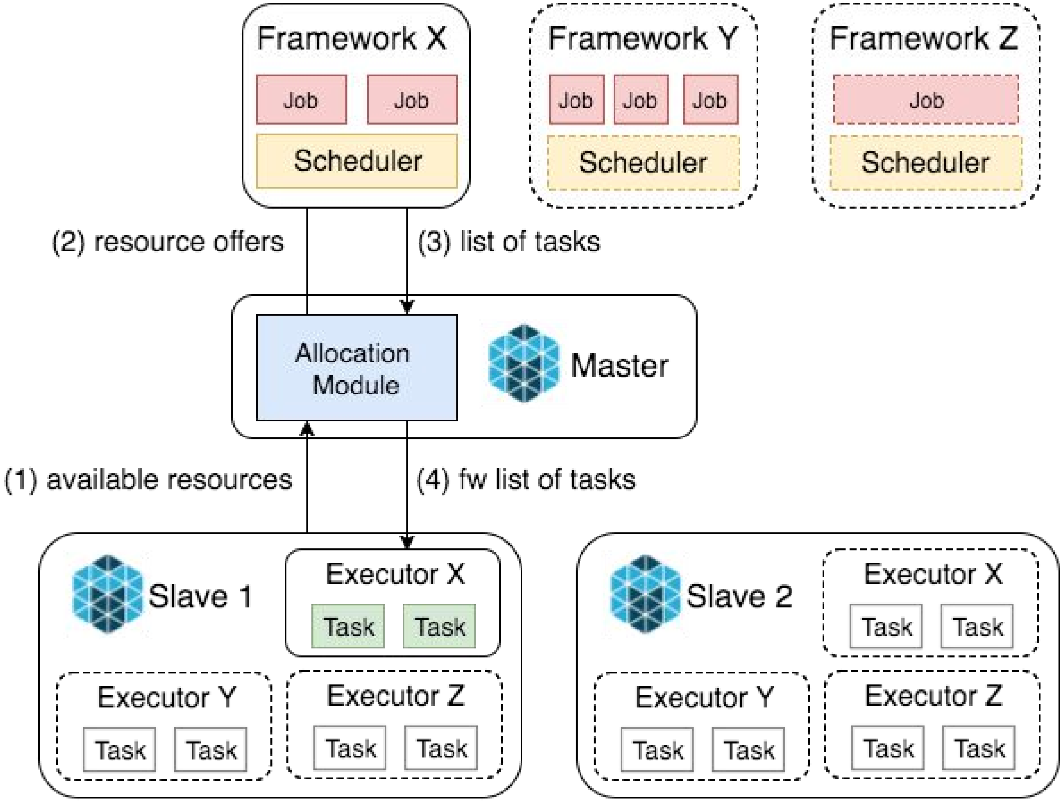
source: [Omega: flexible, scalable schedulers for large compute clusters]

Mesos Architecture



- Master
  - a mediator between slave resources and frameworks
  - enables fine-grained sharing of resources by making resource offers
- Slave
  - manages resources on physical node and runs executors
- Framework
  - application that solves a specific use case
  - *Scheduler* negotiates with master and handles resource offers
  - *Executors* consume resources and run tasks on slaves

Two-Level Scheduling



- Slave nodes report to Master amount of available resources
- Allocation module starts offering resources to frameworks
- Framework receives offers
  - if resources do not satisfy its needs - rejects the offer
  - if resources satisfy its demands - creates list of *tasks* and sends to master
- Master verifies tasks and forwards to executor (and launches the executor if it's not running)

Resource offer

```
id: { value: "0cb2328a-61c2-4316-91ef-cbbb6ebbf504-O1" }
framework_id: { value: "0cb2328a-61c2-4316-91ef-cbbb6ebbf504-0001" }
slave_id: { value: "0cb2328a-61c2-4316-91ef-cbbb6ebbf504-S0" }
hostname: "mesos-slave"
resources { name: "cpus", type: SCALAR, scalar { value: 6.0 }, role: "*" }
resources { name: "mem", type: SCALAR, scalar { value: 6762.0 }, role: "*" }
resources { name: "disk", type: SCALAR, scalar { value: 13483.0 }, role: "*" }
resources { name: "ports", type: RANGES, ranges { range { begin: 31000, end: 32000 } }, role: "*" }
url {
  scheme: "http"
  address {
    hostname: "mesos-slave"
    ip: "172.18.0.5"
    port: 5151
  }
  path: "/slave(1)"
}
```

Framework Scheduler

```scala
class SomeMesosScheduler extends Scheduler {
  override def resourceOffers(driver: SchedulerDriver, offers: List[Offer]): Unit = {
    for(offer <- offers){
      stateLock.synchronized {
        if(isOfferValid(offer)){
          val executorInfo = buildExecutorInfo(driver, "Executor A"))
          //amount of tasks is calculated to fully use resources from the offer
          val tasks = buildTasks(offer, executorInfo)
          driver.launchTasks(List(offer.getId), tasks)
        } else {
          driver.declineOffer(offer.getId)
        }
      }
    }
  }
  //rest of the methods implementations go here
}
```

Dominant Resource Fairness (DRF)

- *Dominant resource*
  - a resource of specific type (cpu, ram, etc.) which is most demanded by a framework among other resources it needs
  - the resource is identified as a share of the total cluster resources of the same type

- *Dominant share*
  - a share of dominant resource allocated to a framework in the cluster

- Example:
  - Cluster total: 9 CPU & 18 GB RAM
  - Framework A tasks need < 3 CPU, 1 GB > (or < 33% CPU, 5% RAM >)
  - Framework B tasks need < 1 CPU, 4 GB > (or < 11% CPU, 22% RAM >)

- DRF algorithm computes frameworks’ dominant shares and tries to maximize the smallest dominant share in the system

DRF Demo

- 3 frameworks with < *8% CPU, 7.5% RAM* > demand each

Name	Active Tasks	CPUs	Mem	Disk	Max Share
Framework C	4	2	2.0 GB	0 B	33.333%
Framework B	4	2	2.0 GB	0 B	33.333%
Framework A	4	2	2.0 GB	0 B	33.333%

Name	Active Tasks	CPUs	Mem	Disk	Max Share
Framework C	4	2	2.0 GB	0 B	33.333%
Framework B	5	2.5	2.5 GB	0 B	41.667%
Framework A	3	1.5	1.5 GB	0 B	25%

- Framework A < *33% CPU, 15% RAM* >, Framework B < *16% CPU, 30% RAM* >)

Name	Active Tasks	CPUs	Mem	Disk	Max Share
Framework A	2	4	2.0 GB	0 B	66.667%
Framework B	2	2	3.9 GB	0 B	59.154%

- Framework A < *33% CPU, 15% RAM* >, Framework B < *16% CPU, 36% RAM* >)

Name	Active Tasks	CPUs	Mem	Disk	Max Share
Framework A	2	4	2.0 GB	0 B	66.667%
Framework B	1	1	2.4 GB	0 B	36.971%

Name	Active Tasks	CPUs	Mem	Disk	Max Share
Framework A	1	2	1000 MB	0 B	33.333%
Framework B	2	2	4.9 GB	0 B	73.943%

DRF properties

- Sharing incentive
  - Each user should be better off sharing the cluster, than exclusively using her own partition of the cluster. Consider a cluster with identical nodes and n users. Then a user should not be able to allocate more tasks in a cluster partition consisting of 1/n of all resources.
- Strategy-proofness
  - Users should not be able to benefit by lying about their resource demands. This provides incentive compatibility, as a user cannot improve her allocation by lying.
- Envy-freeness
  - A user should not prefer the allocation of another user. This property embodies the notion of fairness.
- Pareto efficiency
  - It should not be possible to increase the allocation of a user without decreasing the allocation of at least another user. This property is important as it leads to maximizing system utilization subject to satisfying the other properties.

source: [Dominant Resource Fairness: Fair Allocation of Multiple Resource Types]()

Resource Reservation

- Goals:
  - allocate all single slave resources to one type of framework
  - divide cluster between several framework types or organisations
  - framework groups prioritization and guaranteed allocation
- Static reservation
  - slave node is configured on start (cannot be reserved for another role or unreserved)

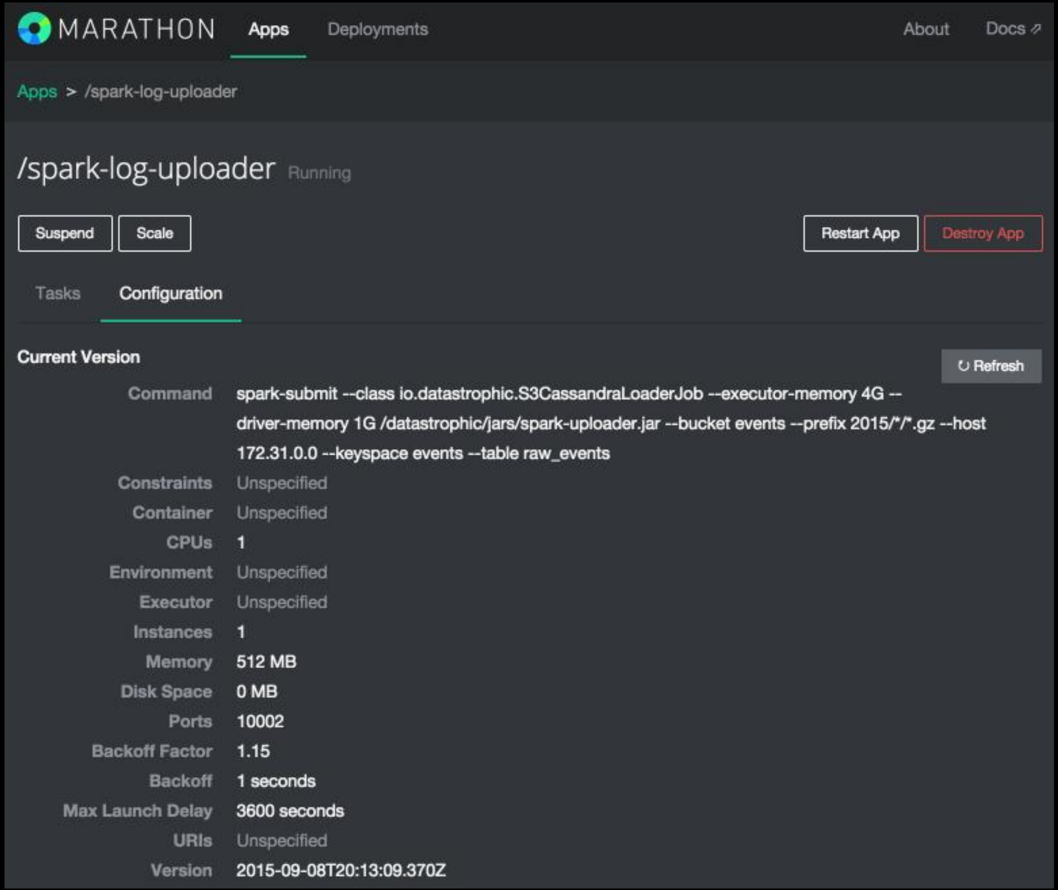
    `--resources="cpus:4;mem:2048;cpus(spark):8;mem(spark):4096"`
- Dynamic reservation
  - resources are reserved/unreserved within a respond to resource offer

    `Offer::Operation::Reserve`
  - MESOS-2018
- Extras:
  - persistent volumes
  - multiple disk resources

Resource Isolation

- Goals:
  - running tasks isolation and capping of runtime resources
  - programmatic control over task resources
  - use images to allow different environments
- Docker containerizer
  - executed tasks are docker containers (e.g. microservices packed in Docker)
- Mesos containerizer (default)
  - Mesos-native (no dependencies on other technologies)
  - provides fine-grained controls (cgroups/namespaces)
  - provides disk usage limits controls
- Composing
  - allows using multiple containerizers together
  - the first containerizer supporting task configuration will be used to launch it

Ubiquitous frameworks: Marathon

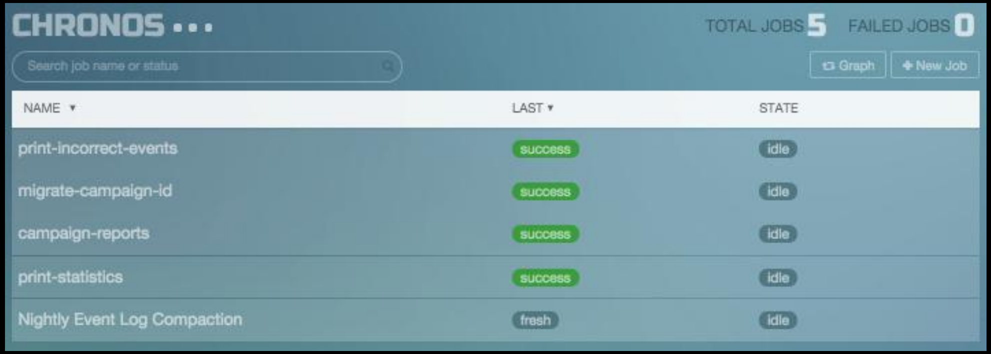


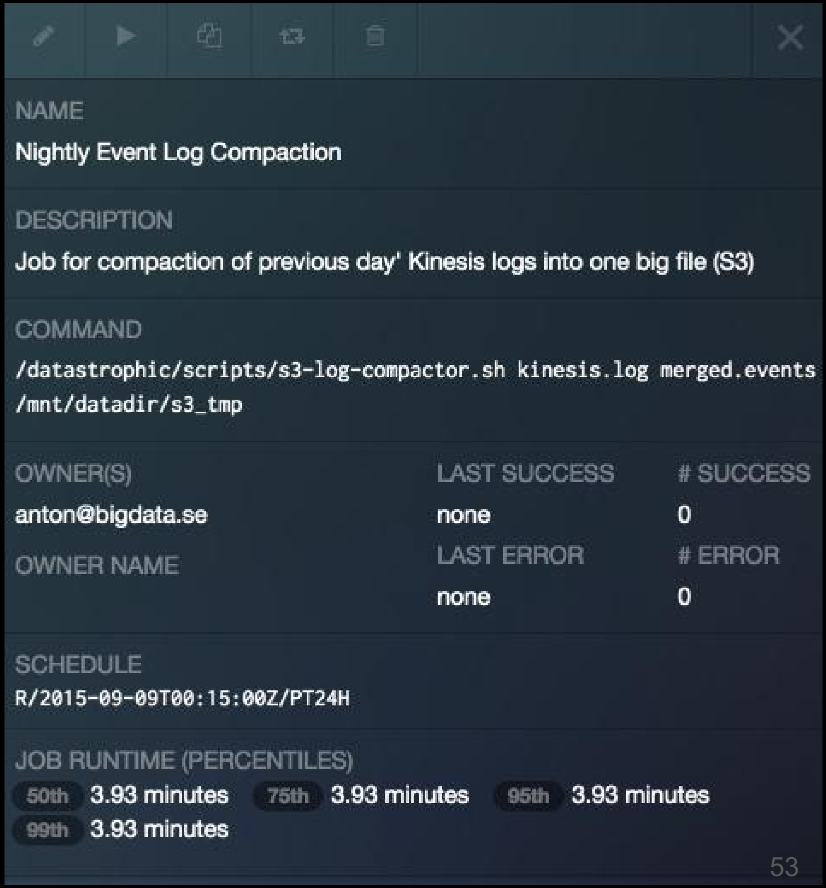
- distributed init.d
- long running tasks execution
- HA mode with ZooKeeper
- Docker executor
- REST API

Marathon: launching Chronos in Docker

```
curl -XPOST 'http://marathon:8080/v2/apps' -H 'Content-Type: application/json' -d '{
  "id": "chronos",
  "container": {
    "type": "DOCKER",
    "docker": {
      "network": "HOST",
        "image": "datastrophic/chronos:mesos-0.27.1-chronos-2.5",
        "parameters": [
             { "key": "env", "value": "CHRONOS_HTTP_PORT=4400" },
             { "key": "env", "value": "CHRONOS_MASTER=zk://zookeeper:2181/mesos" },
             { "key": "env", "value": "CHRONOS_ZK_HOSTS=zookeeper:2181"}
        ]
    }
  },
  "ports": [ 4400 ],
  "cpus": 1,
  "mem": 512,
  "instances": 1
}'
```

Ubiquitous frameworks: Chronos



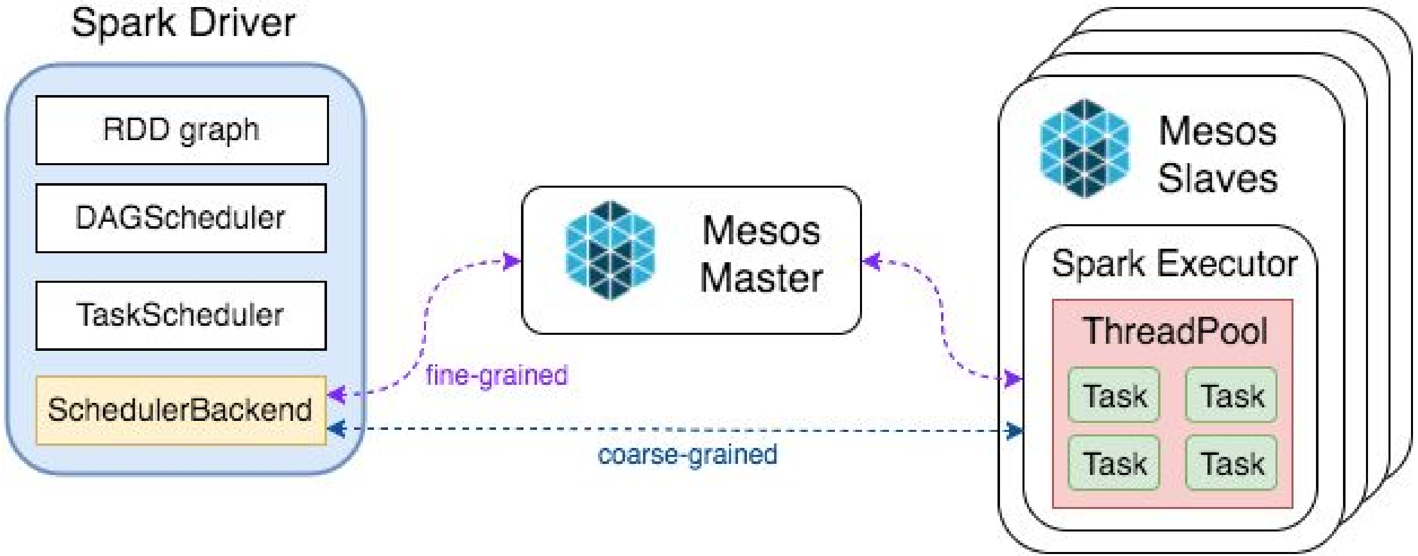


- distributed cron
- HA mode with ZooKeeper
- supports graphs of jobs
- sensitive to network failures

More Mesos frameworks

- Spark
- Hadoop
- Cassandra
- Kafka
- Myriad: YARN on Mesos
- Storm
- Samza

Spark on Mesos



- Coarse-grained mode(default)
  - Spark Executor is launched one per Slave and acquires all available cores in cluster
  - Tasks are scheduled by Spark relying on its RPC mechanism (Akka)
- Fine-grained mode
  - Spark Executor is launched one per Slave with minimal resources needed (1 core)
  - Spark tasks are executed as Mesos tasks and use Mesos semantics

Spark on Mesos

- Coarse-grained mode

```
/opt/spark/bin/spark-submit \
    --class io.datastrophic.demo.SparkJob \
    --master mesos://zk://zookeeper:2181/mesos \
    --conf "spark.cores.max=10" \
    /opt/jobs/spark-jobs-assembly.jar
```

- Fine-grained mode

```
/opt/spark/bin/spark-submit \
    --class io.datastrophic.demo.SparkJob \
    --master mesos://zk://zookeeper:2181/mesos \
    --conf "spark.mesos.coarse=false"\
    /opt/jobs/spark-jobs-assembly.jar
```

Spark on Mesos vs. YARN

- Mesos (coarse-grained)

```
/opt/spark/bin/spark-submit \
        --class io.datastrophic.demo.SparkJob \
        --master mesos://zk://zookeeper:2181/mesos \
        --conf "spark.cores.max=100" \
        /opt/jobs/spark-jobs-assembly.jar
```

- YARN

```
/opt/spark/bin/spark-submit \
        --class io.datastrophic.demo.SparkJob \
         --master yarn \
         --num-executors 25 \
         --executor-cores 4 \
        /opt/jobs/spark-jobs-assembly.jar
```

Running Spark via Marathon

```
curl -XPOST 'http://marathon:8080/v2/apps' -H 'Content-Type: application/json' -d '{
    "cmd": "/opt/spark/bin/spark-submit
    --class io.datastrophic.demo.SparkJob
    --master mesos://zk://zookeeper:2181/mesos
    --deploy-mode client
    /opt/jobs/spark-jobs-assembly.jar",
    "id": "spark-pi",
    "cpus": 1,
    "mem": 1024,
    "instances": 1
}'
```

Running Spark via Chronos

```
curl -L -H 'Content-Type: application/json' -X POST http://mesos:4400/scheduler/iso8601 -d '{
    "name": "Scheduled Spark Submit Job",
    "/opt/spark/bin/spark-submit
    --class io.datastrophic.demo.SparkJob
    --master mesos://zk://zookeeper:2181/mesos
    /opt/jobs/spark-jobs-assembly.jar",
    "shell": true,
    "async": false,
    "cpus": 0.1,
    "disk": 256,
    "mem": 1024,
    "owner": "anton@datastrophic.io",
    "description": "Spark Job executed every 3 minutes",
    "schedule": "R/2016-03-14T12:35:00.000Z/PT3M"
}'
```

Spark deployment strategies

- Binaries distribution
  - every node in the cluster must have Spark libraries installed in the same locations
  - pros: easy to start with
  - cons: hard to upgrade, hard to have several Spark versions simultaneously
- Edge nodes
  - use nodes with specific environment setup which are reachable from Mesos cluster and keep Spark executor jars in accessible location like S3, HTTP or HDFS
  - pros: easy to use multiple Spark versions, minimal dependencies on Mesos
  - cons: hard to maintain in case of multi-tenancy
- Dockerized environment
  - Instead of edge nodes use Docker containers with environment configured for specific needs (hosts still need to be reachable from Mesos cluster) and use Docker Spark executor
  - pros: highly isolated environments for specific needs, could be upgraded independently, zero impact on cluster nodes
  - cons: could be hard to properly setup and configure

Mesos Framework Walkthrough

- ***Throttler***
  - a demo framework for load testing Cassandra
  - load intensity is controlled by parameters: total queries, queries per task and parallelism (how many Mesos tasks to run in parallel)
- Goals
  - take a look at working (simple) Mesos application
  - see how Scheduler, Executor and framework launcher could be implemented
- Sources:
  - source code and dockerized Mesos cluster configuration are available at github/datastrophic/mesos-workshop
  - all the examples (and even more) available as well

Questions

@antonkirillov datastrophic.io