

SMACK Architectures

Building data processing platforms with
Spark, Mesos, Akka, Cassandra and Kafka

Anton Kirillov

Big Data AW Meetup
Sep 2015

Who is this guy?

- Scala programmer
 - Focused on distributed systems
 - Building data platforms with SMACK/Hadoop
 - Ph.D. in Computer Science
-
- Big Data engineer/consultant at Big Data AB
 - Currently at Ooyala Stockholm (Videoplaza AB)
 - Working with startups

Roadmap

- SMACK stack overview
- Storage layer layout
- Fixing NoSQL limitations
- Cluster resource management
- Reliable scheduling and execution
- Data ingestion options
- Preparing for failures

SMACK Stack



- **Spark** - fast and general engine for distributed, large-scale data processing



- **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

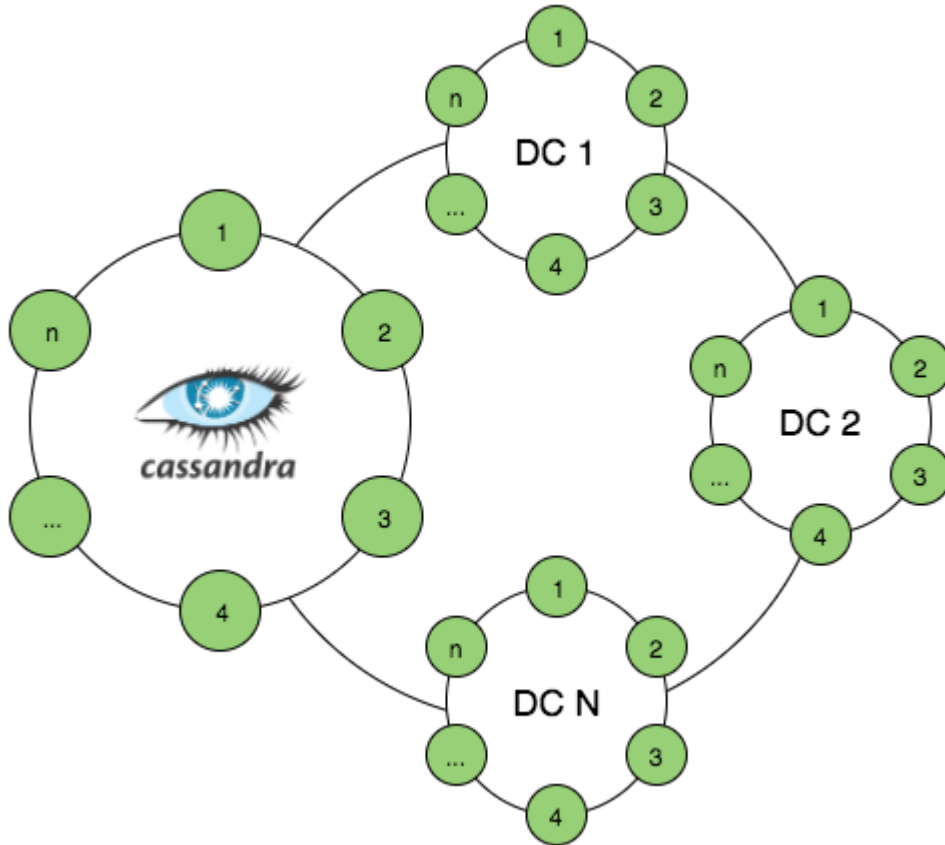


- **Cassandra** - distributed, highly available database designed to handle large amounts of data across multiple datacenters



- **Kafka** - a high-throughput, low-latency distributed messaging system designed for handling real-time data feeds

Storage Layer: Cassandra



- optimized for heavy write loads
- configurable CA (CAP)
- linearly scalable
- XDCR support
- easy cluster resizing and inter-DC data migration

Cassandra Data Model

```
CREATE TABLE campaign(  
  id uuid,  
  year int,  
  month int,  
  day int,  
  views bigint,  
  clicks bigint,  
  PRIMARY KEY (id, year, month, day)  
);
```

- nested sorted map
- should be optimized for read queries
- data is distributed across nodes by partition key

```
INSERT INTO campaign(id, year, month, day, views, clicks)  
VALUES(40b08953-a...,2015, 9, 10, 1000, 42);
```

```
SELECT views, clicks FROM campaign  
WHERE id=40b08953-a... and year=2015 and month>8;
```

Spark/Cassandra Example

```
CREATE TABLE event(  
  id uuid,  
  ad_id uuid,  
  campaign uuid,  
  ts bigint,  
  type text,  
  PRIMARY KEY(id)  
);
```

- calculate total views per campaign for given month for all campaigns

```
val sc = new SparkContext(conf)  
  
case class Event(id: UUID, ad_id: UUID, campaign: UUID, ts: Long, `type`: String)  
  
sc.cassandraTable[Event]("keyspace", "event")  
  .filter(e => e.`type` == "view" && checkMonth(e.ts))  
  .map(e => (e.campaign, 1))  
  .reduceByKey(_ + _)  
  .collect()
```

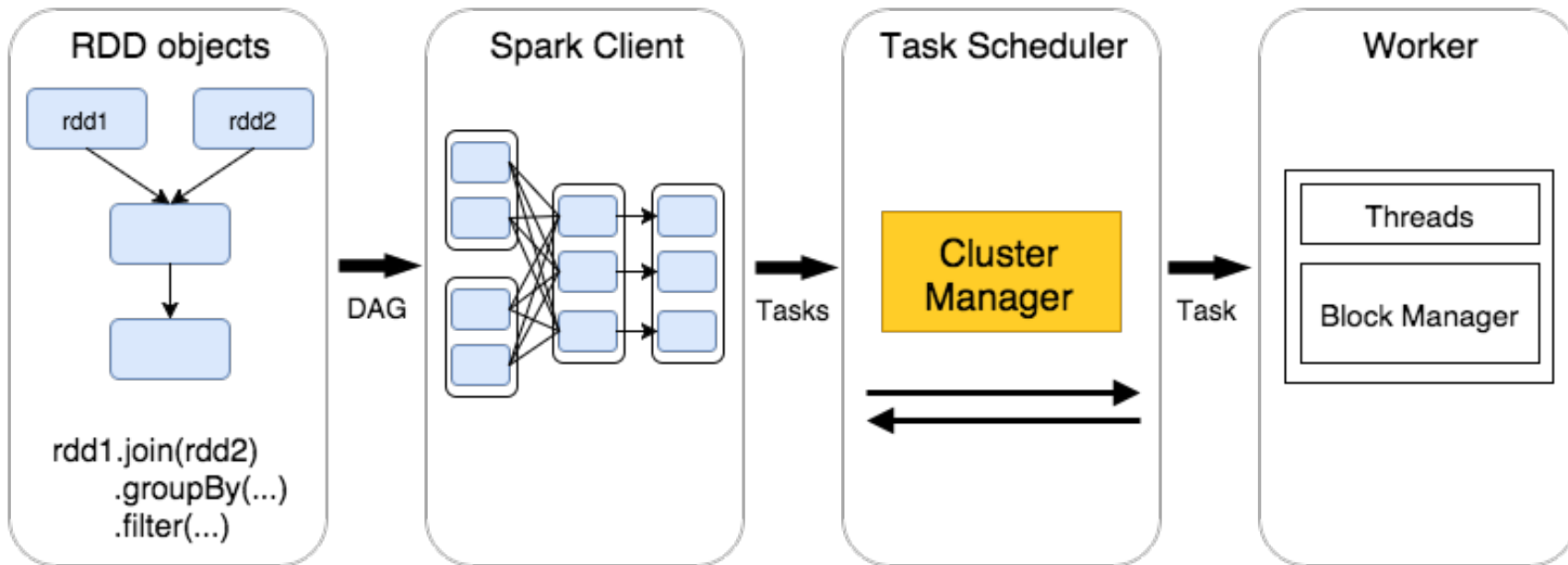
Naive Lambda example with Spark SQL

```
case class CampaignReport(id: String, views: Long, clicks: Long)

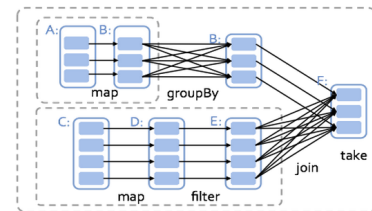
sql("""SELECT campaign.id as id, campaign.views as views,
        campaign.clicks as clicks, event.type as type
        FROM campaign
        JOIN event ON campaign.id = event.campaign
        """).rdd
.groupBy(row => row.getAs[String]("id"))
.map{ case (id, rows) =>
  val views = rows.head.getAs[Long]("views")
  val clicks = rows.head.getAs[Long]("clicks")

  val res = rows.groupBy(row => row.getAs[String]("type")).mapValues(_.size)
  CampaignReport(id, views = views + res("view"), clicks = clicks + res("click"))
}.saveToCassandra("keyspace", "campaign_report")
```

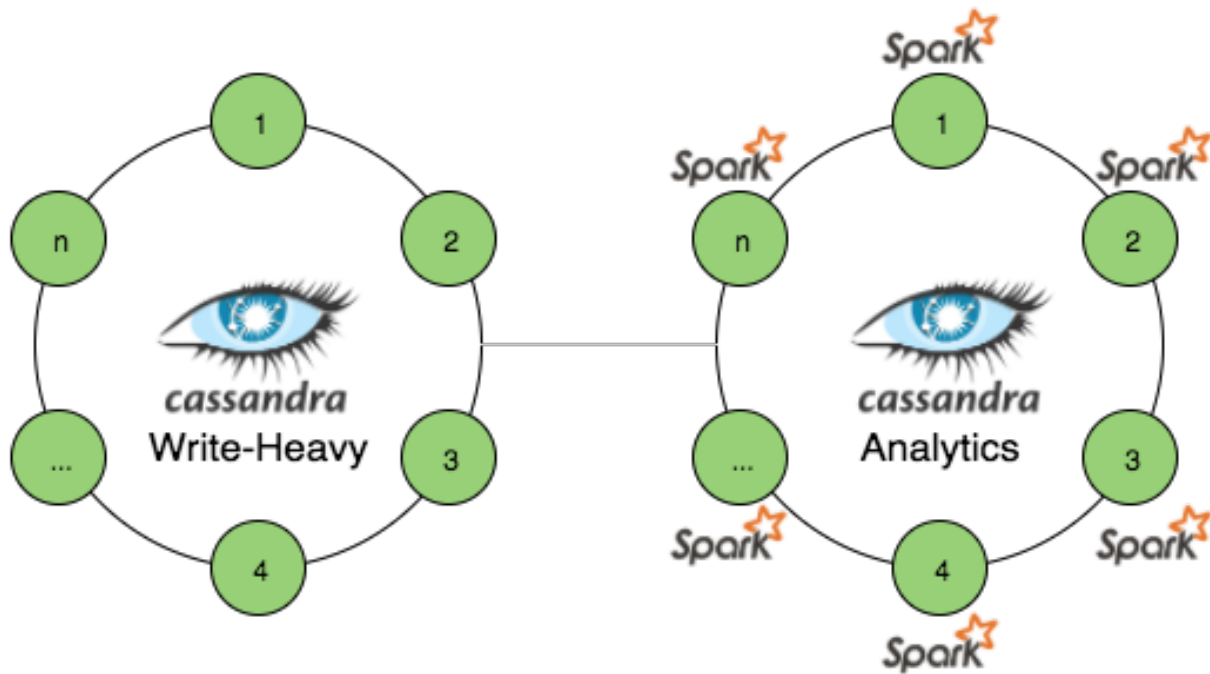

Let's take a step back: Spark Basics



- RDD operations(transformations and actions) form DAG
- DAG is split into stages of tasks which are then submitted to cluster manager
- stages combine tasks which don't require shuffling/repartitioning
- tasks run on workers and results then return to client



Architecture of Spark/Cassandra Clusters

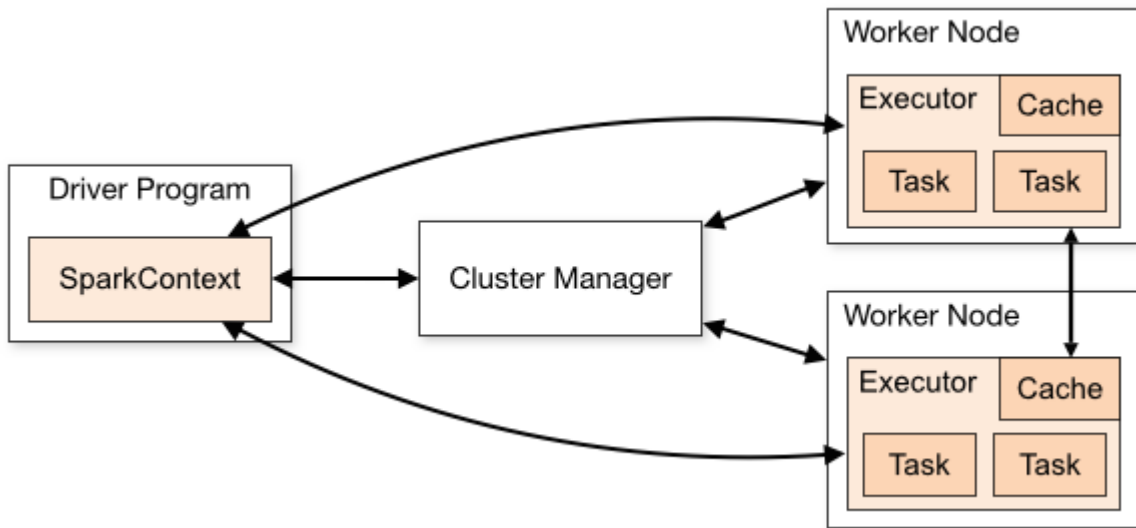


Separate Write & Analytics:

- 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

Spark Applications Deployment Revisited



Cluster Manager:

- Spark Standalone
- YARN
- Mesos

Managing Cluster Resources: Mesos

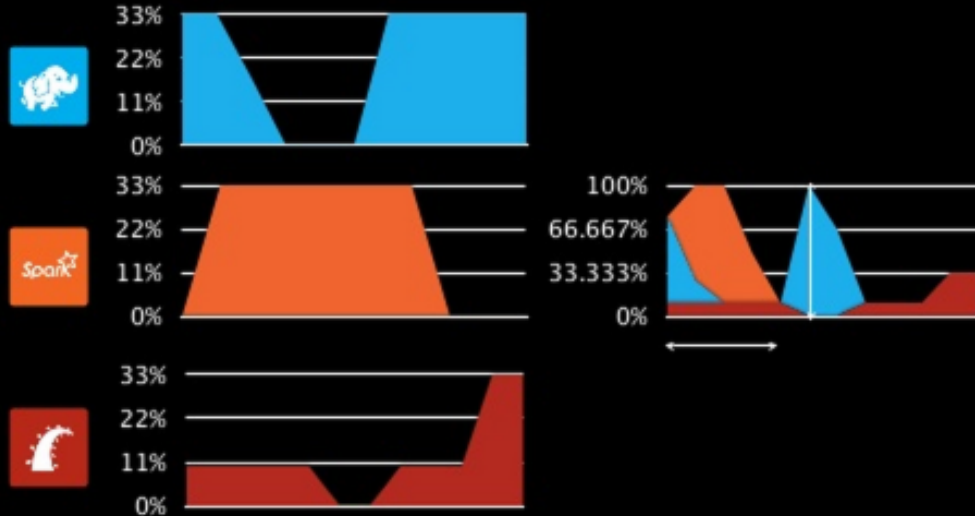
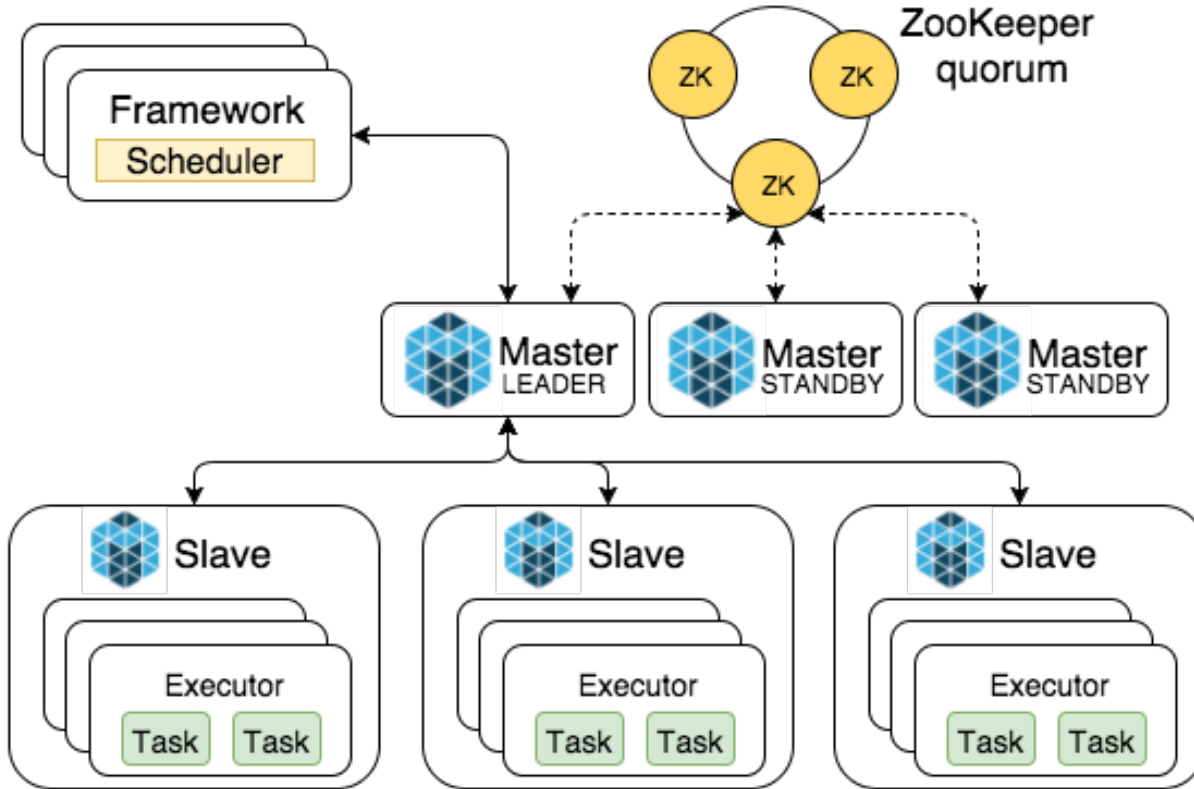


image source: <http://www.slideshare.net/caniszczyk/apache-mesos-at-twitter-texas-linuxfest-2014>

- heterogenous workloads
- full cluster utilization
- static vs. dynamic resource allocation
- fault tolerance and disaster recovery
- single resource view at datacenter level

Mesos Architecture Overview

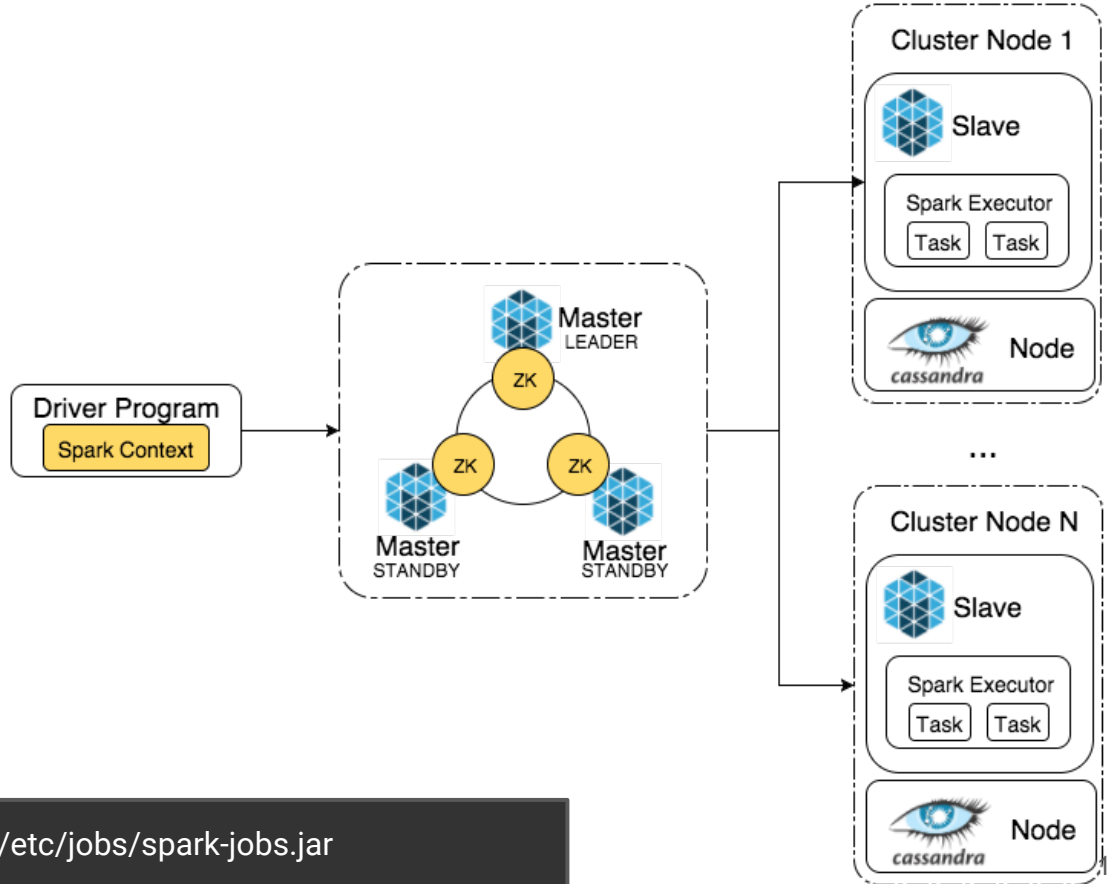


- leader election and service discovery via ZooKeeper
- slaves publish available resources to master
- master sends resource offers to frameworks
- scheduler replies with tasks and resources needed per task
- master sends tasks to slaves

Bringing Spark, Mesos and Cassandra Together

Deployment example

- Mesos Masters and ZooKeepers collocated
- Mesos Slaves and Cassandra nodes collocated to enforce better data locality for Spark
- Spark binaries deployed to all worker nodes and spark-env is configured
- Spark Executor JAR uploaded to S3



Invocation example

```
spark-submit --class io.datastrophic.SparkJob /etc/jobs/spark-jobs.jar
```

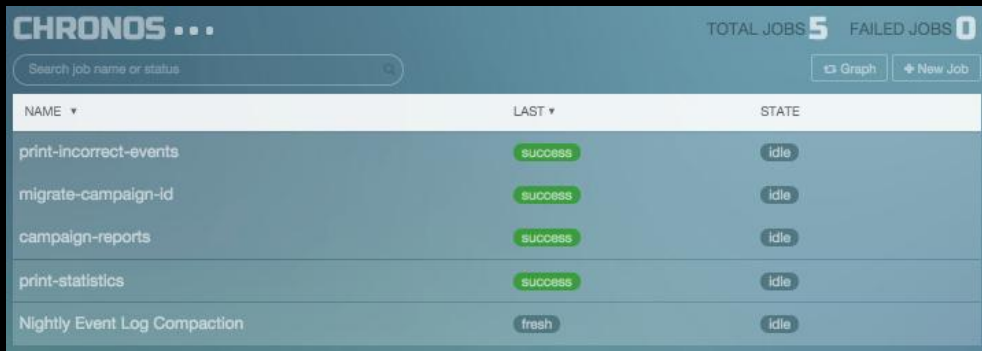
Marathon

The screenshot displays the Marathon web interface. At the top, there is a navigation bar with the Marathon logo, 'MARATHON', and tabs for 'Apps' (selected), 'Deployments', 'About', and 'Docs'. Below the navigation bar, the breadcrumb path is 'Apps > /spark-log-uploader'. The main content area shows the application name '/spark-log-uploader' with a 'Running' status. There are four buttons: 'Suspend', 'Scale', 'Restart App', and 'Destroy App'. Below the buttons, there are two tabs: 'Tasks' and 'Configuration' (selected). The 'Configuration' tab shows the 'Current Version' of the application. A 'Refresh' button is located to the right of the 'Current Version' header. The configuration details are as follows:

Command	spark-submit --class io.datastrophic.S3CassandraLoaderJob --executor-memory 4G --driver-memory 1G /datastrophic/jars/spark-uploader.jar --bucket events --prefix 2015/**.gz --host 172.31.0.0 --keyspace events --table raw_events
Constraints	Unspecified
Container	Unspecified
CPU	1
Environment	Unspecified
Executor	Unspecified
Instances	1
Memory	512 MB
Disk Space	0 MB
Ports	10002
Backoff Factor	1.15
Backoff	1 seconds
Max Launch Delay	3600 seconds
URIs	Unspecified
Version	2015-09-08T20:13:09.370Z

- long running tasks execution
- HA mode with ZooKeeper
- Docker executor
- REST API

Chronos



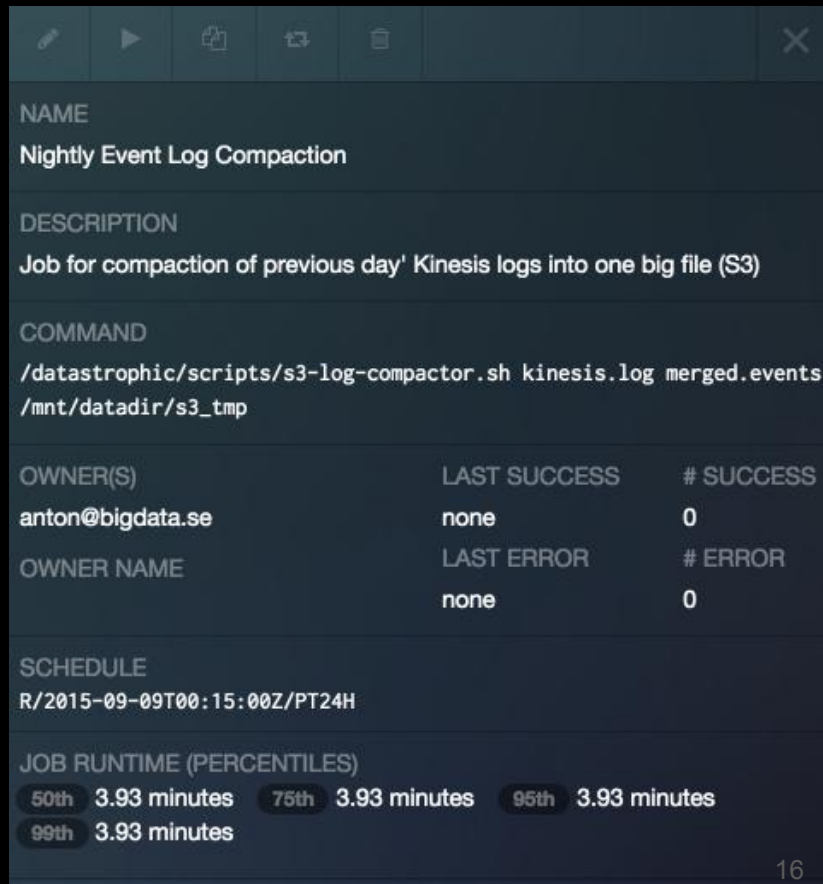
CHRONOS ... TOTAL JOBS 5 FAILED JOBS 0

Search job name or status

Graph New Job

NAME	LAST	STATE
print-incorrect-events	success	idle
migrate-campaign-id	success	idle
campaign-reports	success	idle
print-statistics	success	idle
Nightly Event Log Compaction	fresh	idle

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



NAME
Nightly Event Log Compaction

DESCRIPTION
Job for compaction of previous day' Kinesis logs into one big file (S3)

COMMAND
`/datastrophic/scripts/s3-log-compactor.sh kinesis.log merged.events /mnt/datadir/s3_tmp`

OWNER(S)	LAST SUCCESS	# SUCCESS
anton@bigdata.se	none	0

OWNER NAME	LAST ERROR	# ERROR
	none	0

SCHEDULE
R/2015-09-09T00:15:00Z/PT24H

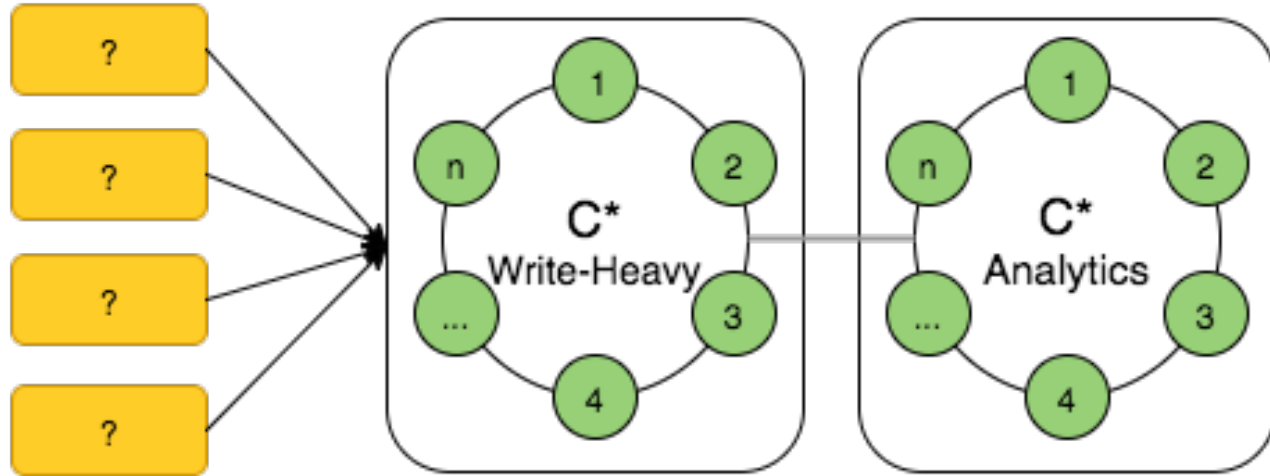
JOB RUNTIME (PERCENTILES)

50th	3.93 minutes	75th	3.93 minutes	95th	3.93 minutes
99th	3.93 minutes				

More Mesos frameworks

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

Data ingestion: endpoints to consume the data



Endpoint requirements:

- high throughput
- resiliency
- easy scalability
- back pressure

Akka features

```
class JsonParserActor extends Actor {
  def receive = {
    case s: String => Try(Json.parse(s).as[Event]) match {
      case Failure(ex) => log.error(ex)
      case Success(event) => sender ! event
    }
  }
}

class HttpActor extends Actor {
  def receive = {
    case req: HttpRequest =>
      system.actorOf(Props[JsonParserActor]) ! req.body
    case e: Event =>
      system.actorOf(Props[CassandraWriterActor]) ! e
  }
}
```

- actor model implementation for JVM
- message-based and asynchronous
- no shared mutable state
- easy scalability from one process to cluster of machines
- actor hierarchies with parental supervision
- not only concurrency framework:
 - akka-http
 - akka-streams
 - akka-persistence

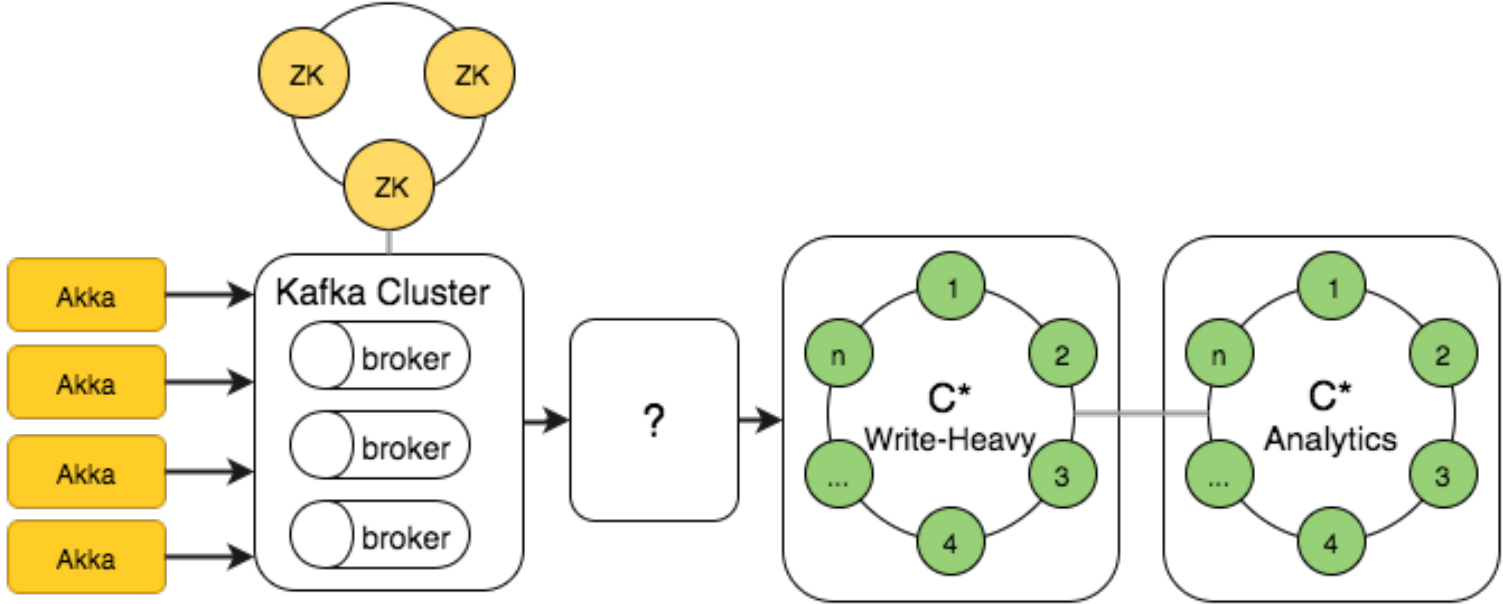
Writing to Cassandra with Akka

```
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 ! WriteSuccessful  
      }  
    }  
  }  
}
```

Cassandra meets Batch Processing

- writing raw data (events) to Cassandra with Akka is easy
- but computation time of aggregations/rollups will grow with amount of data
- Cassandra is still designed for fast serving but not batch processing, so pre-aggregation of incoming data is needed
- actors are not suitable for performing aggregation due to stateless design model
- micro-batches partially solve the problem
- reliable storage for raw data is still needed

Kafka: distributed commit log



- pre-aggregation of incoming data
- consumers read data in batches
- available as Kinesis on AWS

Publishing to Kafka with Akka Http

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

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))
          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"))
}
```

Spark Streaming



- variety of data sources
- at-least-once semantics
- exactly-once semantics available with Kafka Direct and idempotent storage



Spark Streaming: Kinesis example

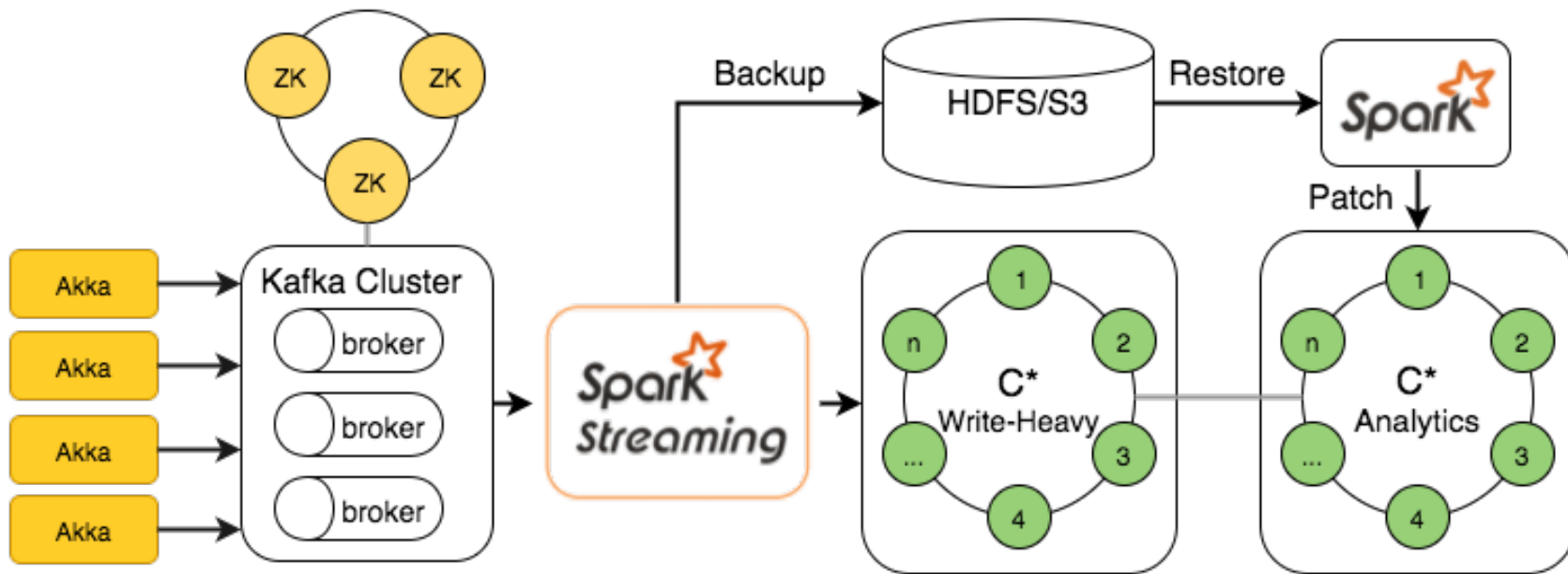
```
val ssc = new StreamingContext(conf, Seconds(10))

val kinesisStream = KinesisUtils.createStream(ssc, appName, streamName,
    endpointURL, regionName, InitialPositionInStream.LATEST,
    Duration(checkpointInterval), StorageLevel.MEMORY_ONLY)
}

//transforming given stream to Event and saving to C*
kinesisStream.map(JsonUtils.byteArrayToEvent)
                .saveToCassandra(keyspace, table)

ssc.start()
ssc.awaitTermination()
```

Designing for Failure: Backups and Patching



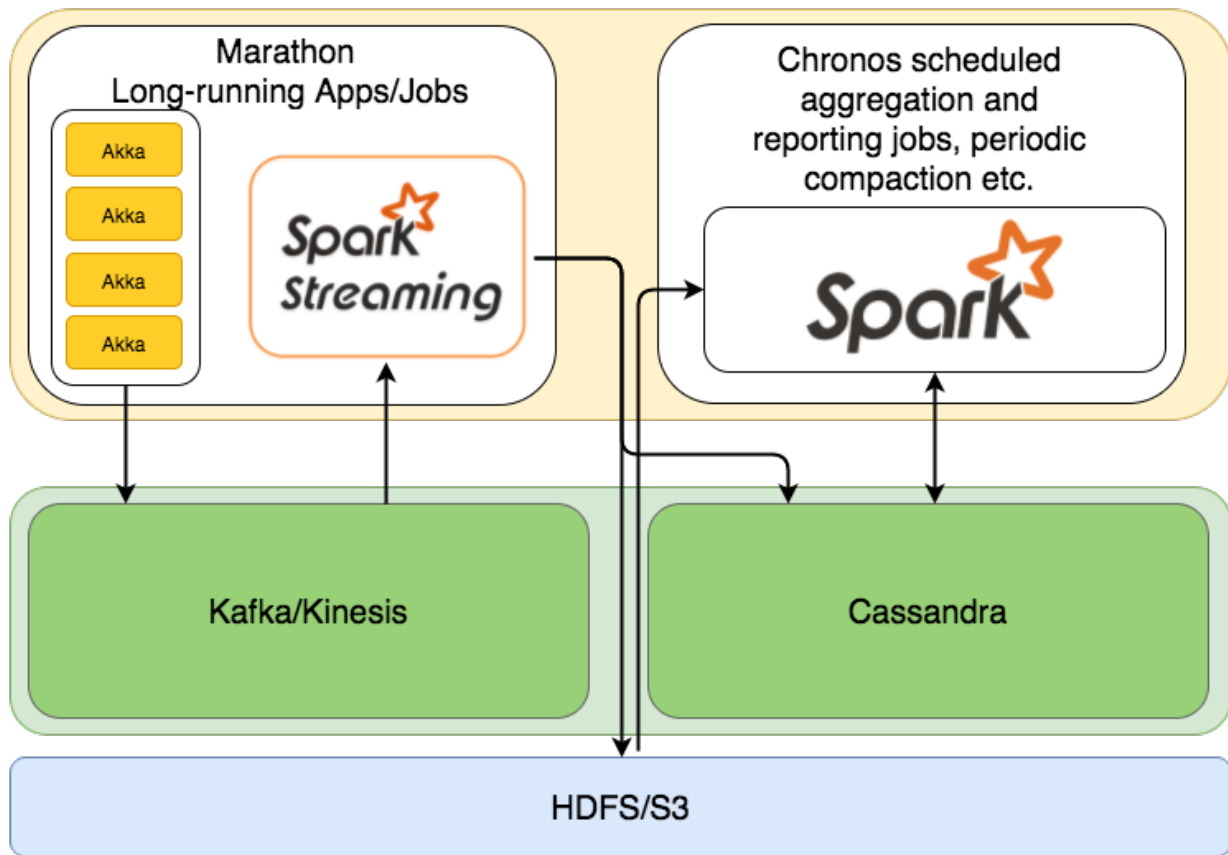
- be prepared for failures and broken data
- design backup and patching strategies upfront
- idempotency should be enforced

Restoring backup from S3

```
val sc = new SparkContext(conf)

sc.textFile(s"s3n://bucket/2015/*/*.gz")
  .map(s => Try(JsonUtils.stringToEvent(s)))
  .filter(_.isSuccess).map(_.get)
  .saveToCassandra(config.keyspace, config.table)
```

The big picture



So what SMACK is

- 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)

Questions

[@antonkirillov](#)

[datastrophic.io](#)