Lambda Architecture with Spark Streaming, Kafka, Cassandra, Akka, Scala

Helena Edelson
@helenaedelson

Who Is This Person?

DATASTAX

- Spark Cassandra Connector committer
- Akka contributor - 2 new features in Akka Cluster
- Big Data & Scala conference speaker
- Currently Sr Software Engineer, Analytics @ DataStax
- Sr Cloud Engineer, VMware,CrowdStrike,SpringSource…
- Prev Spring committer - Spring AMQP, Spring Integration

Talk Roadmap

DATASTAX

What Lambda Architecture & Delivering Meaning

Why Spark, Kafka, Cassandra & Akka integration

How Composable Pipelines - Code

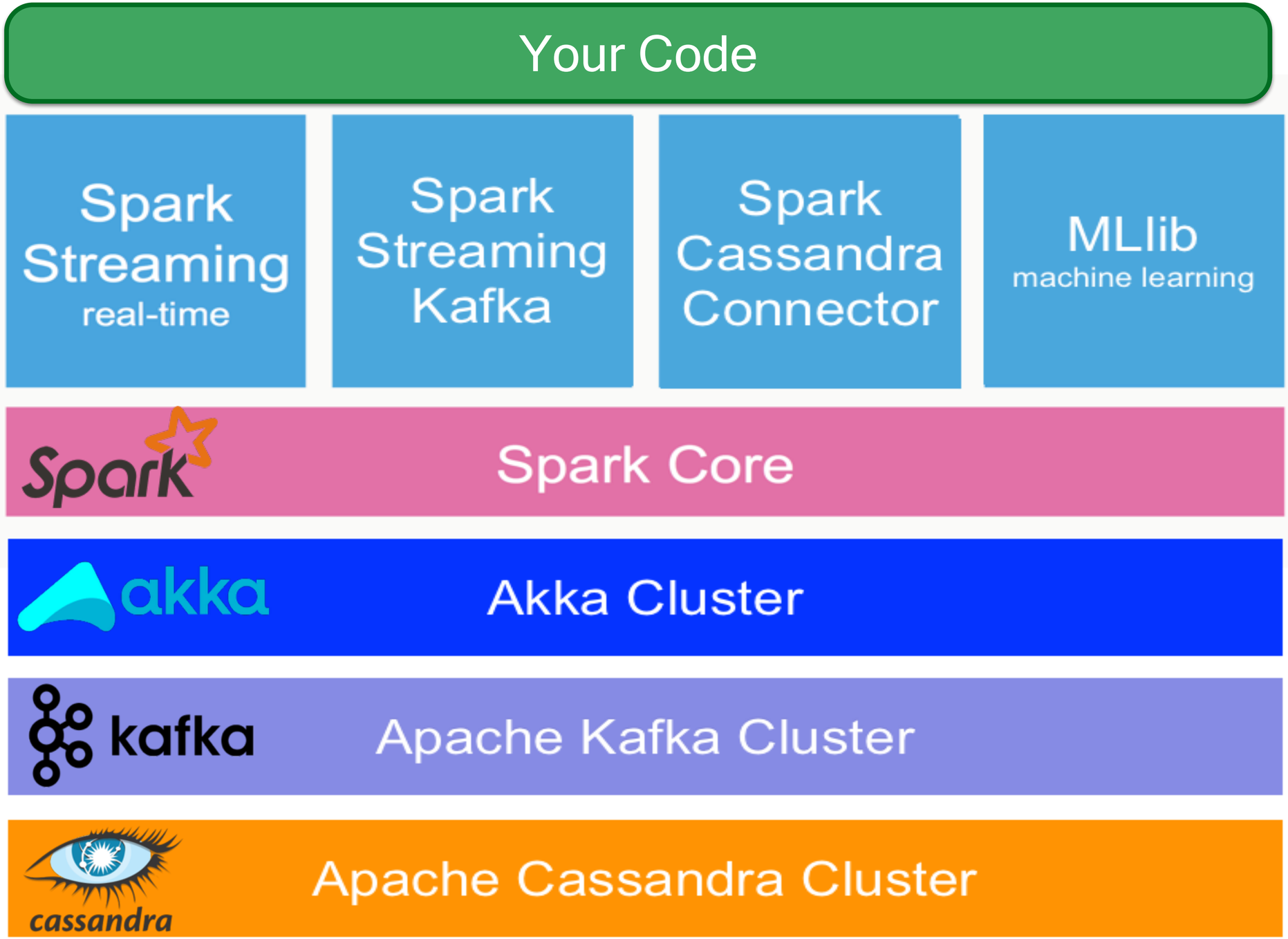
helena.edelson@datastax.com

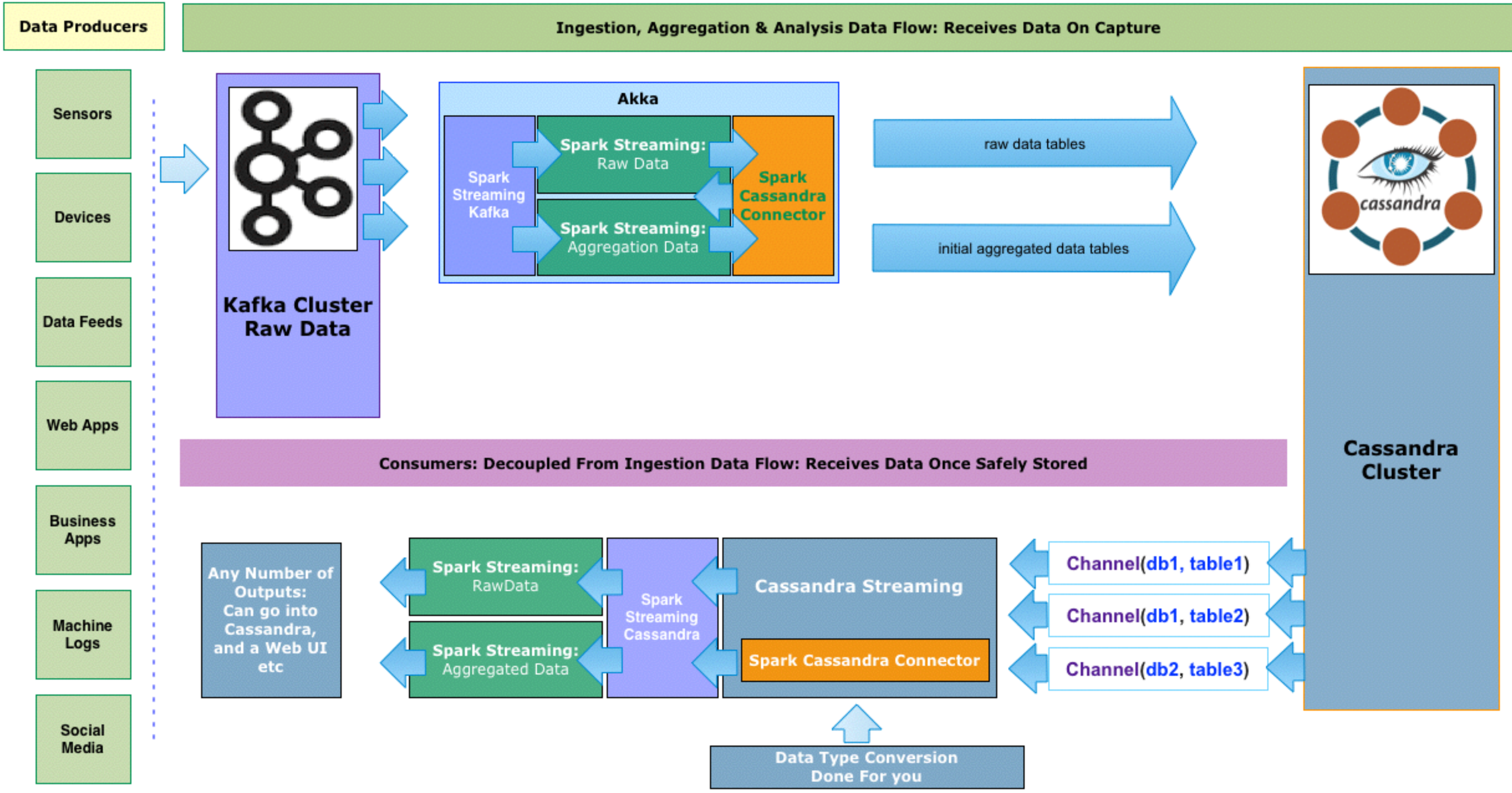
I need fast access to historical data on the fly for predictive modeling with real time data from the stream

Lambda Architecture

A data-processing architecture designed to handle *massive quantities* of data by taking advantage of both batch and stream processing methods.

- Spark is one of the few data processing frameworks that allows you to seamlessly integrate batch and stream processing
  - Of petabytes of data
  - In the same application


Your Code
Spark Streaming
real-time
Spark Streaming Kafka
Spark Cassandra Connector
MLlib
machine learning
Spark
Spark Core
akka
Akka Cluster
kafka
Apache Kafka Cluster
cassandra
Apache Cassandra Cluster


Data Producers
Ingestion, Aggregation & Analysis Data Flow: Receives Data On Capture
Sensors
Devices
Data Feeds
Web Apps
Business Apps
Machine Logs
Social Media
Kafka Cluster Raw Data
Akka
Spark Streaming Kafka
Spark Streaming: Raw Data
Spark Streaming: Aggregation Data
Spark Cassandra Connector
raw data tables
initial aggregated data tables
cassandra
Cassandra Cluster
Consumers: Decoupled From Ingestion Data Flow: Receives Data Once Safely Stored
Any Number of Outputs: Can go into Cassandra, and a Web UI etc
Spark Streaming: RawData
Spark Streaming: Aggregated Data
Spark Streaming Cassandra
Cassandra Streaming
Spark Cassandra Connector
Channel(db1, table1)
Channel(db1, table2)
Channel(db2, table3)
Data Type Conversion Done For you

This repository Search Explore Gist Blog Help helena

killrweather / killrweather

Unwatch 69 Star 361 Fork 73

KillrWeather is a reference application (work in progress) showing how to easily achieve Lambda Architecture with Apache Spark Streaming, Apache Cassandra, Apache Kafka and Akka for fast, streaming computations on time series data in asynchronous event-driven environments. — Edit

167 commits 3 branches 0 releases 4 contributors

branch: master killrweather / +

Minor use case addition.

helena authored 19 seconds ago latest commit 1f1988cf68

data	Adding much bigger data file	7 months ago
diagrams	Version upgrades.	7 days ago
killrweather-app/src	Version upgrades for: Spark, Connector, Akka, Akka Streams and Scala …	14 hours ago
killrweather-clients/src/main	Version upgrades for: Spark, Connector, Akka, Akka Streams and Scala …	14 hours ago
killrweather-core/src/main	Version upgrades for: Spark, Connector, Akka, Akka Streams and Scala …	14 hours ago
killrweather-examples/src/main	Added new logback configs and finalized log levels.	3 months ago
project	Version upgrades for: Spark, Connector, Akka, Akka Streams and Scala …	14 hours ago
sigar	Added Sigar for test metrics collection via Akka Cluster, handled IT …	8 months ago
.gitignore	Adding first diagram.	7 months ago

Code
Issues 6
Pull requests 3
Wiki
Pulse
Graphs
Settings

SSH clone URL
git@github.com:killrweath

Moving Data Between Systems Is

~~*Difficult*~~ *Risky and Expensive*

How Do We Approach This?

Strategies

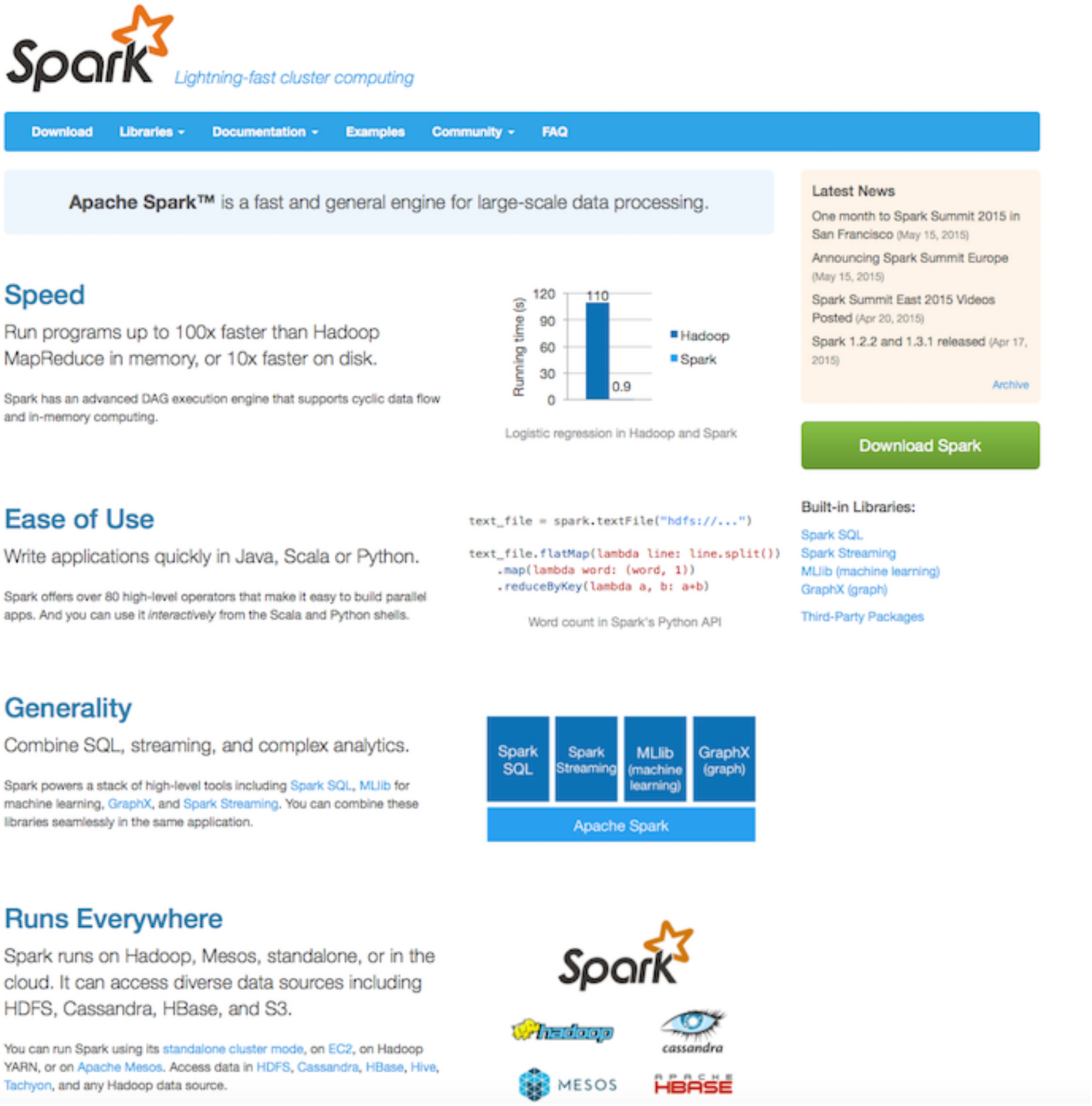
DATASTAX

- Scalable Infrastructure
- Partition For Scale
- Replicate For Resiliency
- Share Nothing
- Asynchronous Message Passing
- Parallelism
- Isolation
- Data Locality
- Location Transparency

My Nerdy Chart

DATASTAX

Strategy	Technologies
Scalable Infrastructure / Elastic	Spark, Cassandra, Kafka
Partition For Scale, Network Topology Aware	Cassandra, Spark, Kafka, Akka Cluster
Replicate For Resiliency	Spark,Cassandra, Akka Cluster all hash the node ring
Share Nothing, Masterless	Cassandra, Akka Cluster both Dynamo style
Fault Tolerance / No Single Point of Failure	Spark, Cassandra, Kafka
Replay From Any Point Of Failure	Spark, Cassandra, Kafka, Akka + Akka Persistence
Failure Detection	Cassandra, Spark, Akka, Kafka
Consensus & Gossip	Cassandra & Akka Cluster
Parallelism	Spark, Cassandra, Kafka, Akka
Asynchronous Data Passing	Kafka, Akka, Spark
Fast, Low Latency, Data Locality	Cassandra, Spark, Kafka
Location Transparency	Akka, Spark, Cassandra, Kafka



- Fast, distributed, scalable and fault tolerant cluster compute system
- Enables Low-latency with complex analytics
- Developed in 2009 at UC Berkeley AMPLab, open sourced in 2010
- Became an Apache project in February, 2014

kafka

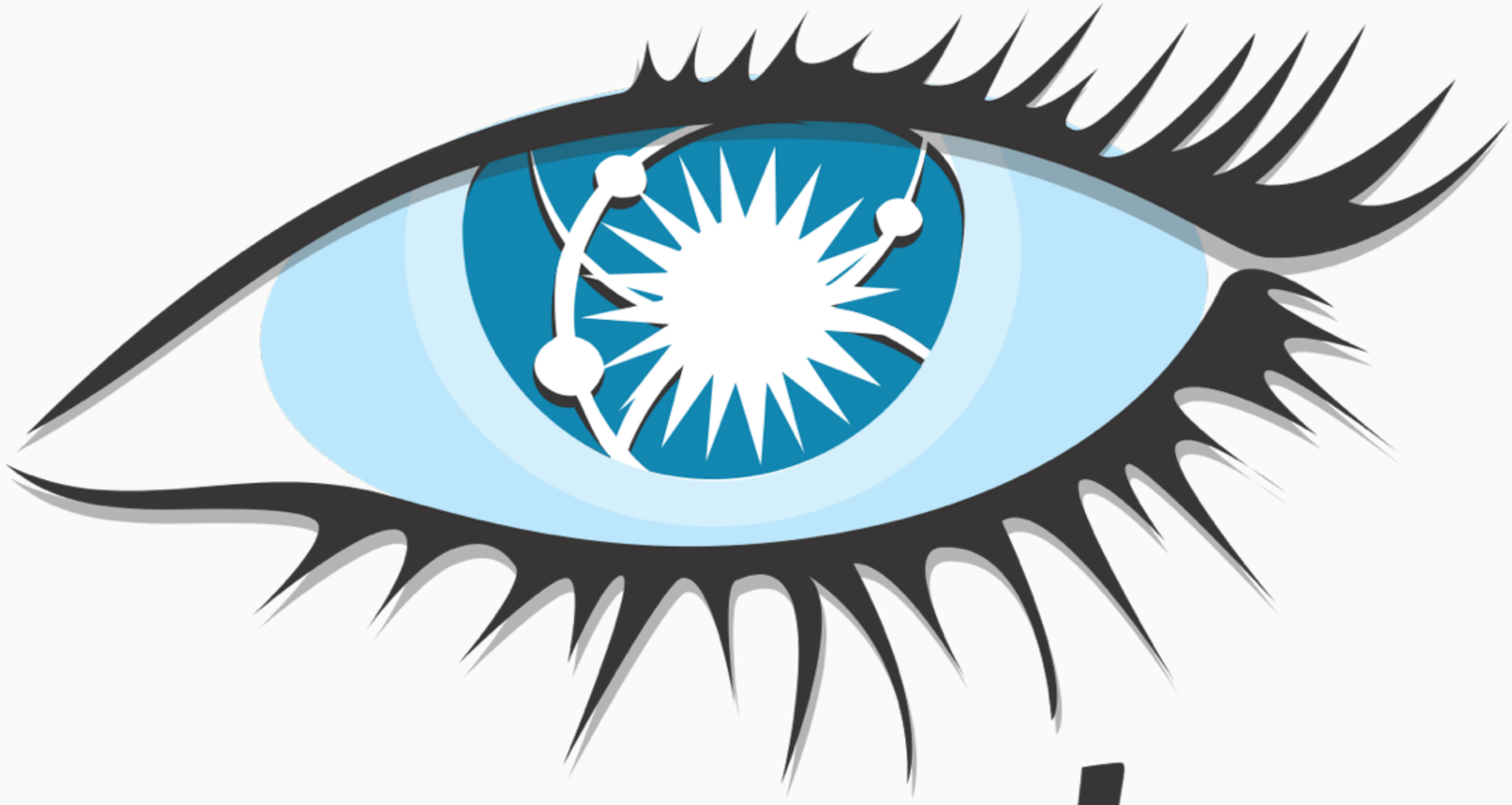
DATASTAX

- High Throughput Distributed Messaging
- Decouples Data Pipelines
- Handles Massive Data Load
- Support Massive Number of Consumers
- Distribution & partitioning across cluster nodes
- Automatic recovery from broker failures

Speaking Of Fault Tolerance…

DATASTAX

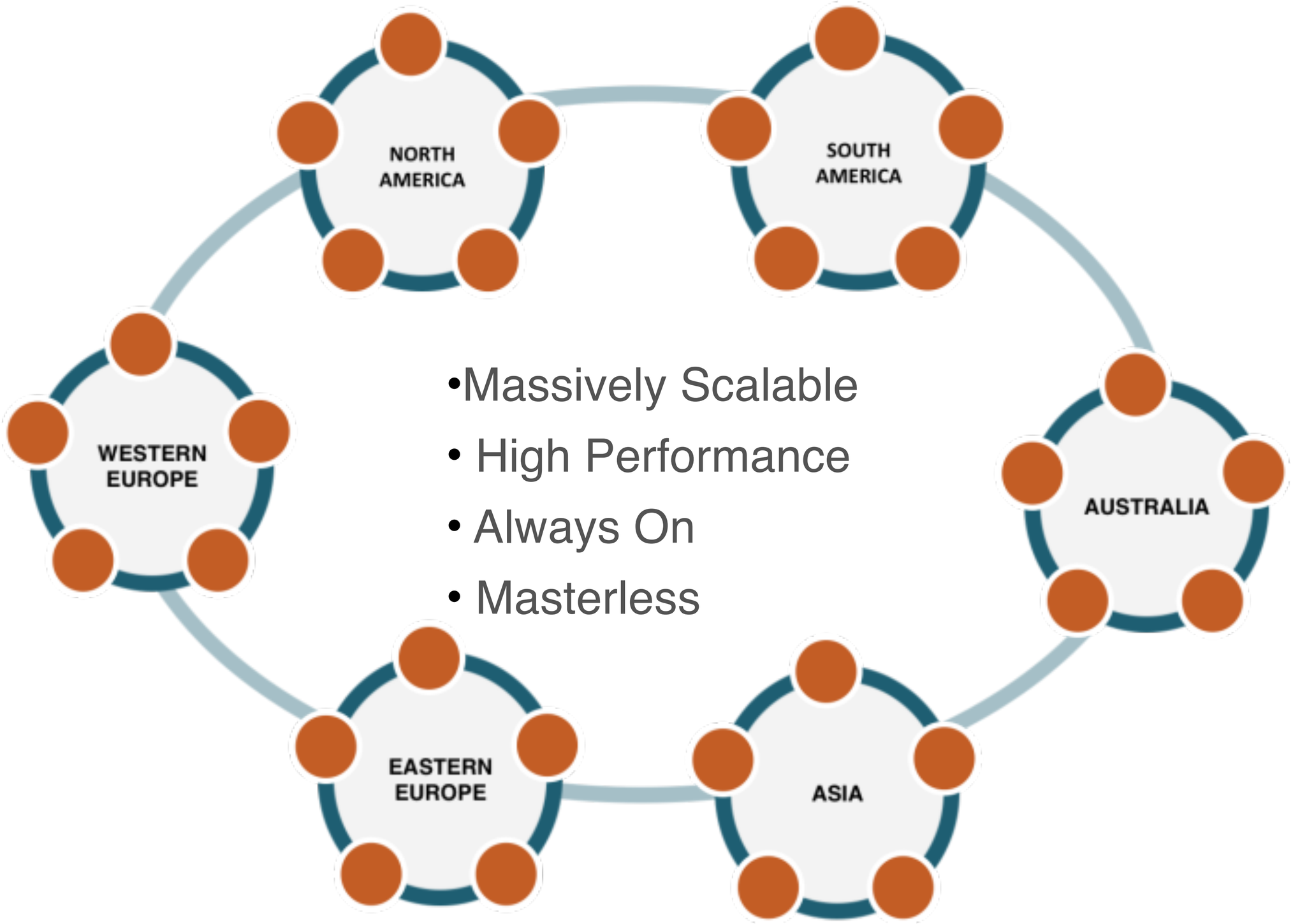
The one thing in your infrastructure you can *always* rely on.



cassandra

Availability

"During Hurricane Sandy, we lost an entire data center. *Completely. Lost. It.* Our data in Cassandra never went offline."

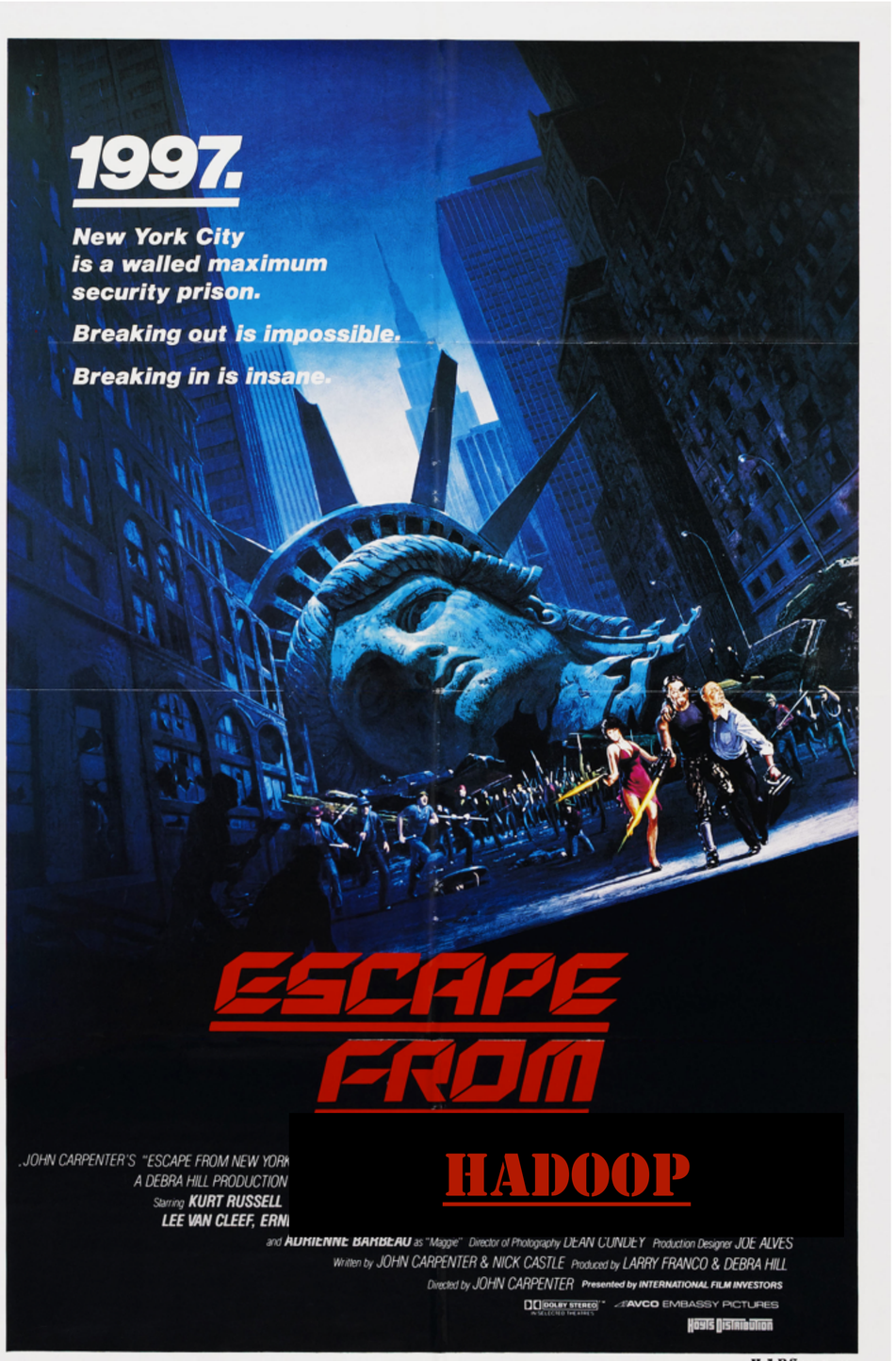

NORTH AMERICA
SOUTH AMERICA
WESTERN EUROPE
AUSTRALIA
EASTERN EUROPE
ASIA
•Massively Scalable
• High Performance
• Always On
• Masterless

- Fault tolerant
  - Hierarchical Supervision
  - Customizable Failure Strategies & Detection
- Asynchronous Data Passing
- Parallelization - Balancing Pool Routers
- Akka Cluster
  - Adaptive / Predictive
  - Load-Balanced Across Cluster Nodes

Scala

I've used Scala with these every single time.

- Stream data from Kafka to Cassandra
- Stream data from Kafka to Spark and write to Cassandra
- *Stream from Cassandra to Spark - coming soon!*
- Read data from Spark/Spark Streaming Source and write to C*
- Read data from Cassandra to Spark



Spark

- Distributed Analytics Platform
- Easy Abstraction for Datasets
- Support in several languages
- Streaming
- Machine Learning
- Graph
- Integrated SQL Queries
- Has Generalized DAG execution

All in one package

And it uses Akka

Most Active OSS In Big Data

DATASTAX

apache / spark
mirrored from git://git.apache.org/spark.git

Watch 427 | Star 2,285 | Fork 2,044

November 11, 2014 – December 11, 2014

Period: **1 month**

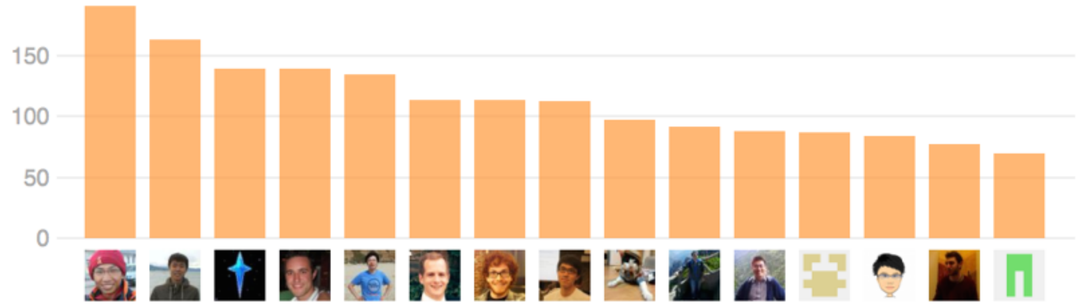
Overview

154 Active Pull Requests

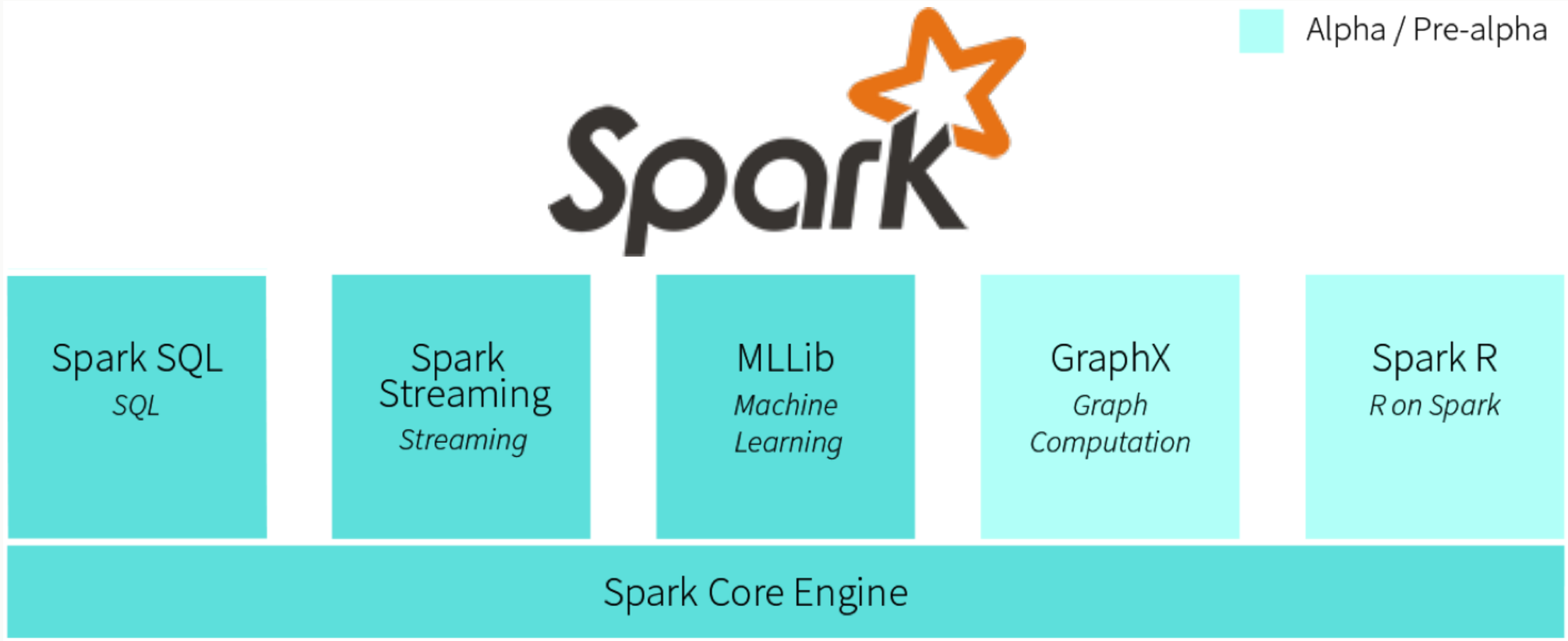
0 Active Issues

0	154	0	0
Merged Pull Requests	Proposed Pull Requests	Closed Issues	New Issues

Excluding merges, **317 authors** have pushed **297 commits** to master and **3,546 commits** to all branches. On master, **702 files** have changed and there have been **38,458 additions** and **18,021 deletions**.



154 Pull requests proposed by **82** people


Alpha / Pre-alpha
Spark
Spark SQL
SQL
Spark Streaming
Streaming
MLLib
Machine Learning
GraphX
Graph Computation
Spark R
R on Spark
Spark Core Engine

Apache Spark - Easy to Use API

DATASTAX

Returns the top (k) highest temps for any location in the year

```
def topK(aggregate: Seq[Double]): Seq[Double] =
  sc.parallelize(aggregate).top(k).collect
```

Returns the top (k) highest temps … in a Future

```
def topK(aggregate: Seq[Double]): Future[Seq[Double]] =
  sc.parallelize(aggregate).top(k).collectAsync
```

Use the Spark Shell to quickly try out code samples

Available in Scala

Spark Shell

```
scala> val rdd3 = rdd2.filter(_ > 5)
rdd3: org.apache.spark.rdd.RDD[Int] = FilteredRDD[5] at filter at <console>:63

scala> rdd3.collect
res6: Array[Int] = Array(6, 8, 10, 12)
```

and python

Pyspark

```
>>> x.collect()
[Row(ckey1=u'Seven', data1=u'Seven', pkey=u'Seven'), Row(ckey1=u'Four', data1=u'Four', pkey=u'Four'), Row(ckey1=u'Th
ree', data1=u'Three', pkey=u'Three'), Row(ckey1=u'Six', data1=u'Six', pkey=u'Six'), Row(ckey1=u'Eight', data1=u'Eigh
t', pkey=u'Eight'), Row(ckey1=u'Five', data1=u'Five', pkey=u'Five'), Row(ckey1=u'New Value', data1=u'Newer Value', p
key=u'One'), Row(ckey1=u'Two', data1=u'Two', pkey=u'Two')]
```

Collection To RDD

DATASTAX

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distributedData = sc.parallelize(data)
distributedData: spark.RDD[Int] =
spark.ParallelCollection@10d13e3e
```

Not Just MapReduce

DATASTAX

def **aggregate**[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp
Aggregate the elements of each partition, and then the results for all th

def **cache**(): RDD.this.type
Persist this RDD with the default storage level (MEMORY_ONLY).

def **cartesian**[U](other: RDD[U])(*implicit* arg0: ClassTag[
Return the Cartesian product of this RDD and another one, that is, the

def **checkpoint**(): Unit
Mark this RDD for checkpointing.

def **coalesce**(numPartitions: Int, shuffle: Boolean = fals
Return a new RDD that is reduced into numPartitions partitions.

def **collect**[U](f: PartialFunction[T, U])(*implicit* arg0:
Return an RDD that contains all matching values by applying f.

def **collect**(): Array[T]
Return an array that contains all of the elements in this RDD.

def **context**: SparkContext
The org.apache.spark.SparkContext that this RDD was created on.

def **count**(): Long
Return the number of elements in the RDD.

def **countApprox**(timeout: Long, confidence: Double = 0.95
Approximate version of count() that returns a potentially incomplete res

def **countApproxDistinct**(relativeSD: Double = 0.05): Long
Return approximate number of distinct elements in the RDD.

def **countApproxDistinct**(p: Int, sp: Int): Long
Return approximate number of distinct elements in the RDD.

def **countByValue**()(*implicit* ord: Ordering[T] = null): Ma
Return the count of each unique value in this RDD as a map of (value,

def **countByValueApprox**(timeout: Long, confidence: Double
Approximate version of countByValue().

def **dependencies**: Seq[Dependency[_]]
Get the list of dependencies of this RDD, taking into account whether t

def **distinct**(): RDD[T]
Return a new RDD containing the distinct elements in this RDD.

def **glom**(): RDD[Array[T]]
Return an RDD created by coalescing all elements within each parti

def **groupBy**[K](f: (T) ⇒ K, p: Partitioner)(*implicit* kt
Return an RDD of grouped items.

def **groupBy**[K](f: (T) ⇒ K, numPartitions: Int)(*implici*
Return an RDD of grouped elements.

def **groupBy**[K](f: (T) ⇒ K)(*implicit* kt: ClassTag[K]):
Return an RDD of grouped items.

val **id**: Int
A unique ID for this RDD (within its SparkContext).

def **intersection**(other: RDD[T], numPartitions: Int): R
Return the intersection of this RDD and another one.

def **intersection**(other: RDD[T], partitioner: Partition
Return the intersection of this RDD and another one.

def **intersection**(other: RDD[T]): RDD[T]
Return the intersection of this RDD and another one.

def **isCheckpointed**: Boolean
Return whether this RDD has been checkpointed or not

def **iterator**(split: Partition, context: TaskContext):
Internal method to this RDD; will read from cache if applicable, or otl

def **keyBy**[K](f: (T) ⇒ K): RDD[(K, T)]
Creates tuples of the elements in this RDD by applying f.

def **map**[U](f: (T) ⇒ U)(*implicit* arg0: ClassTag[U]): RD
Return a new RDD by applying a function to all elements of this RDD

def **mapPartitions**[U](f: (Iterator[T]) ⇒ Iterator[U], p
Return a new RDD by applying a function to each partition of this RD

def **mapPartitionsWithContext**[U](f: (TaskContext, Itera
Return a new RDD by applying a function to each partition of this RD

def **mapPartitionsWithIndex**[U](f: (Int, Iterator[T]) ⇒
Return a new RDD by applying a function to each partition of this RD

def **max**()(*implicit* ord: Ordering[T]): T
Returns the max of this RDD as defined by the implicit Ordering[T].

def **min**()(*implicit* ord: Ordering[T]): T
Returns the min of this RDD as defined by the implicit Ordering[T].

var **name**: String
A friendly name for this RDD

val **partitioner**: Option[Partitioner]
Optionally overridden by subclasses to specify how they are partitioned.

def **partitions**: Array[Partition]
Get the array of partitions of this RDD, taking into account whether the RDD is chec

def **persist**(): RDD.this.type
Persist this RDD with the default storage level (MEMORY_ONLY).

def **persist**(newLevel: StorageLevel): RDD.this.type
Set this RDD's storage level to persist its values across operations after the first tim

def **pipe**(command: Seq[String], env: Map[String, String] = Map(),
Return an RDD created by piping elements to a forked external process.

def **pipe**(command: String, env: Map[String, String]): RDD[String]
Return an RDD created by piping elements to a forked external process.

def **pipe**(command: String): RDD[String]
Return an RDD created by piping elements to a forked external process.

def **preferredLocations**(split: Partition): Seq[String]
Get the preferred locations of a partition (as hostnames), taking into account whethe

def **randomSplit**(weights: Array[Double], seed: Long = Utils.random
Randomly splits this RDD with the provided weights.

def **reduce**(f: (T, T) ⇒ T): T
Reduces the elements of this RDD using the specified commutative and associative

def **repartition**(numPartitions: Int)(*implicit* ord: Ordering[T] = nu
Return a new RDD that has exactly numPartitions partitions.

def **sample**(withReplacement: Boolean, fraction: Double, seed: Long
Return a sampled subset of this RDD.

def **saveAsObjectFile**(path: String): Unit
Save this RDD as a SequenceFile of serialized objects.

def **saveAsTextFile**(path: String, codec: Class[_ <: CompressionCode
Save this RDD as a compressed text file, using string representations of elements.

Spark Basic Word Count

DATASTAX

```
val conf = new SparkConf()
 .setMaster(host).setAppName(app)

val sc = new SparkContext(conf)

sc.textFile(words)
   .flatMap(_.split("\\s+"))
   .map(word => (word.toLowerCase, 1))
   .reduceByKey(_ + _)
   .collect
```

RDDs Can be Generated from a Variety of Sources

Scala Collections

Textfiles

RDD Operations

DATASTAX

```
16  object SparkWordCount extends WordCountBlueprint {
17
18    sc.textFile("./src/main/resources/data/words")
19      .flatMap(_.split("\\s+"))
20      .map(word => (clean(word), 1))
21      .reduceByKey(_ + _)
22      .collect foreach println
23  }
```

Transformation

Action

Setting up C* and Spark

DSE > 4.5.0

Just start your nodes with

`dse cassandra -k`

Apache Cassandra

Follow the excellent guide by Al Tobey

http://tobert.github.io/post/2014-07-15-installing-cassandra-spark-stack.html

When Batch Is Not Enough

Your Data Is Like Candy

DATASTAX

Delicious: you want it now

Your Data Is Like Candy

DATASTAX

Delicious: you want it now

Batch Analytics

Analysis after data has accumulated
Decreases the weight of the data by the time it is processed

Streaming Analytics

Analytics as data arrives.
The data won't be stale and neither will our analytics

Both in same app = Lambda

Spark Streaming

DATASTAX

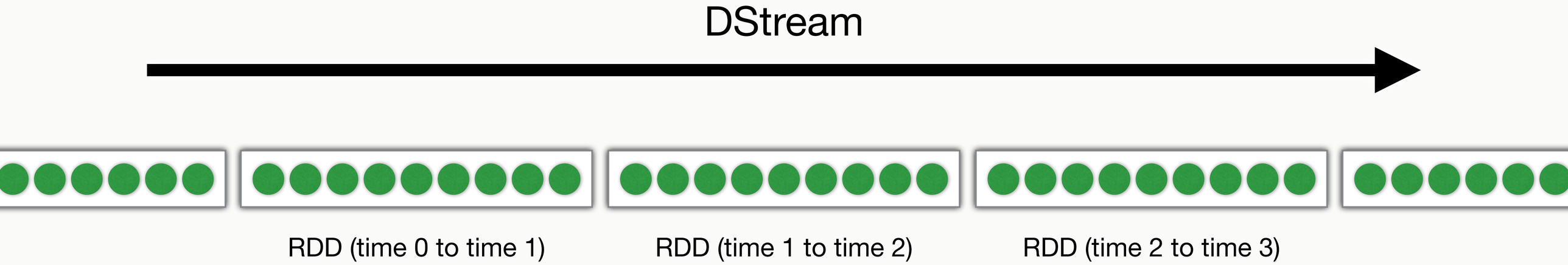
- I want results continuously in the event stream
- I want to run computations in my even-driven async apps
- Exactly once message guarantees

DStream (Discretized Stream)

DATASTAX

Continuous stream of micro batches

- Complex processing models with minimal effort
- Streaming computations on small time intervals



A transformation on a DStream = transformations on its RDDs

Basic Streaming: FileInputDStream

DATASTAX

```
val conf = new SparkConf().setMaster(SparkMaster).setAppName(AppName)
val ssc = new StreamingContext(conf, Milliseconds(500))

ssc.textFileStream("s3n://raw_data_bucket/")
  .flatMap(_.split("\\s+"))
  .map(_.toLowerCase, 1))
  .countByValue()
  .saveToCassandra(keyspace,table)


ssc.checkpoint(checkpointDir)
ssc.start()
ssc.awaitTermination
```

The batch streaming interval

Starts the streaming application piping raw incoming data to a Sink

ReceiverInputDStreams

DATASTAX

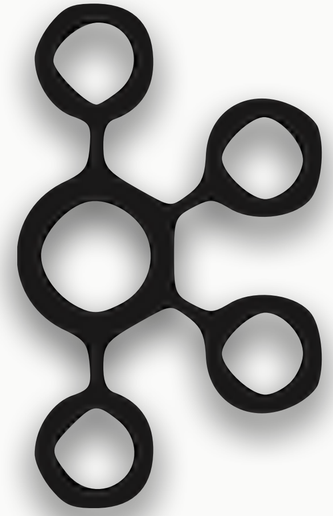
DStreams - the stream of raw data received from streaming sources:

- Basic Source - in the StreamingContext API
- Advanced Source - in external modules and separate Spark artifacts

Receivers

- Reliable Receivers - for data sources supporting acks (like Kafka)
- Unreliable Receivers - for data sources not supporting acks

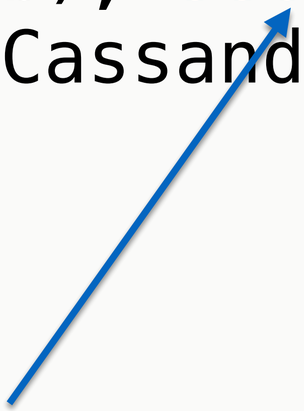
Spark Streaming External Source/Sink



Streaming Window Operations

DATASTAX

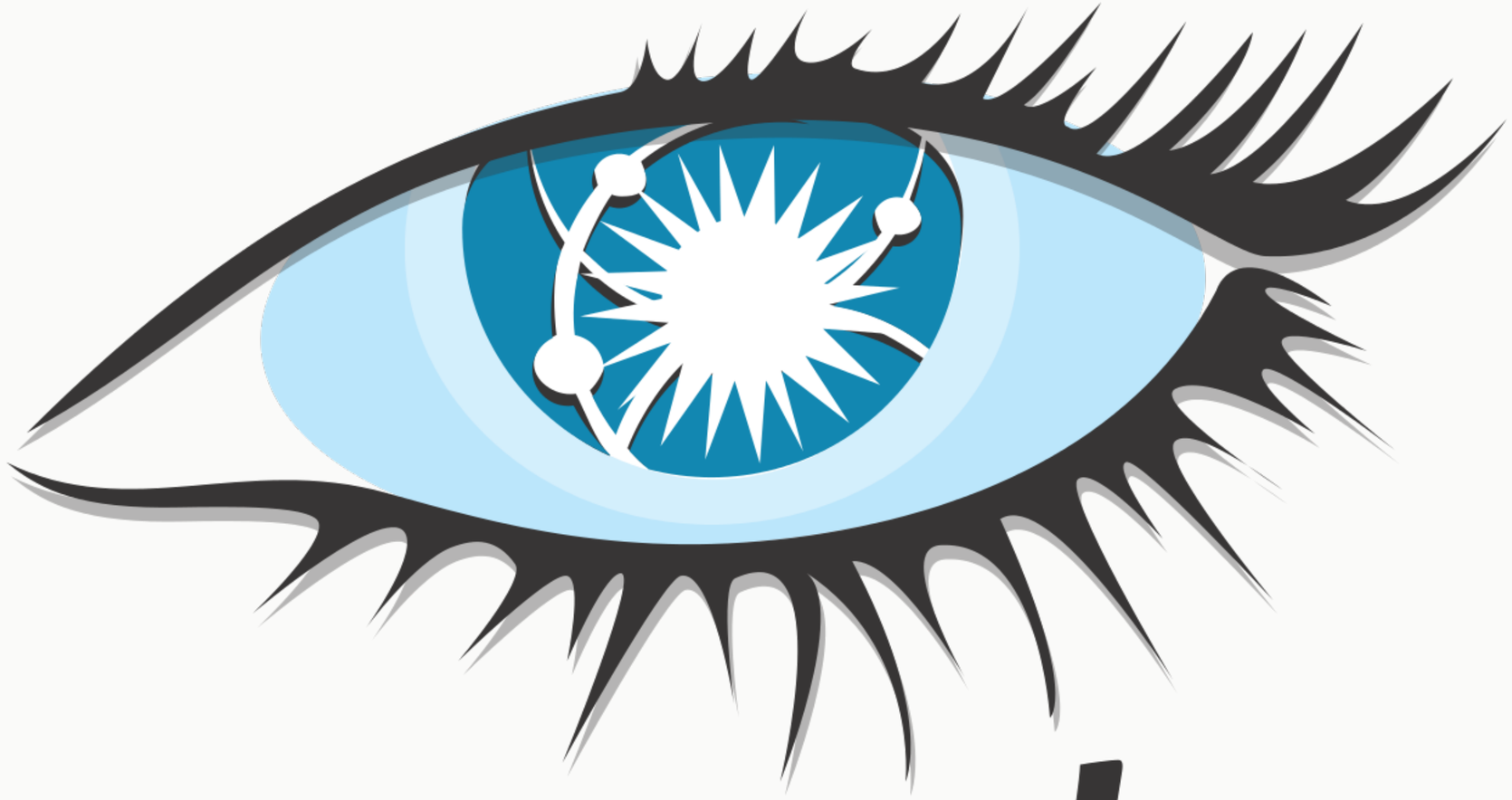
```
kvStream
   .flatMap { case (k,v) => (k,v.value) }
   .reduceByKeyAndWindow((a:Int,b:Int) =>
     (a + b), Seconds(30), Seconds(10))
   .saveToCassandra(keyspace,table)
```



Window Length:
Duration = every 10s

Sliding Interval:
Interval at which the window operation is performed = every 10 s

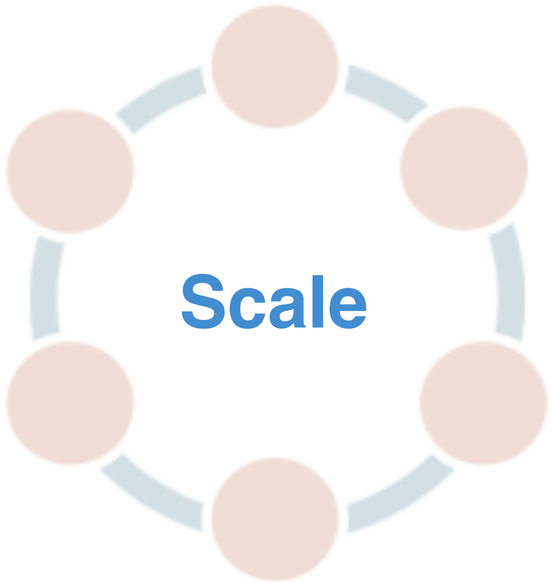
DATASTAX



cassandra


1&1
clemetry
8digits
academicworks
accenture
ACTUATE
Acunu
AddThis
adform
adgear
Adobe
Aeris
AGENTIS
airpim
ALLIED
MYND
ALTOROS
AMARA
ampush
Aol.
apereo
apigee
APIXIO
applifier
AppsFlyer
Apptegic
Arise
ARKIVUM
AURELIUS
POST
avanade
AZUL SYSTEMS
backupify
Barracuda
base
be-mobile
BOXEVER
BROCADE
Burt.
bwin.party
Energy
CABLEVISION
CABOT
CallFire
canary
CANONICAL
Captora
cardSPRING
CARTERA
caserta
charitydic
CharityUSA.com
Chegg
Chill
Cinnober
cisco
cloudkick
CloudPassage
cloudsoft
cloze
CODEMETTLE
COMCAST
comverge
Cornastone
correlor
CrowdRiff
CSR
CURALATE
DachisGroup
DATADIO
Datafiniti
Demandforce
DeNA
despegar.com
digby
Digital Reasoning
Digital River
Disney
DISQUS
docomo
DRILLSTER
druva
dstillery
Dyn
bros
ebay
buddy
gnyte
elasticinbox
eNanoPay
ENGINE
enkata
epsilon
EUROTECH
Eventbrite
exoscale
FACE
FanBridge
feedzai
FINN
fold3
formspring
Frugal Mechanic
FullContact
GAMEFLY
Gazzang
gemalto
GENERAL SENTIMENT
GIGASPACES
GitHub
glassbeam
glispa
Globant
GNIP
GOGRID
GoSquared
GOTiME
GRAINGER
Grid Dynamics
GRIDPOINT
grubHub
GX
Hangtime
has offers
HealthCare
Healthline
Healthx
HOBSONS
HubSpot
hulu
IBM
IMPETUS
inkling
instaclustr
Instagram
Integral
IID
intuit
iovation
ISIDOREY
iSIGHTPARTNERS
ITHAKA
IXONOS
JASPERSOFT
jetpac
Jini
Jumptap
JUNIPER
KASPERSKY
Keen IO
KISSmetrics
KNEWTON
krux
kwarter
kweo
lacuna
LeaseEagle
leaseweb
Libon
librato.
Life360
LIFESTREET
lightt
Lingotek
liquidweb
Lithium
LiveOffice
LIVEPERSON
looplogic
lucid
LYTICS
Mahalo
MARKED UP
martinimedia
Mass Relevance
mate1.com
MediaCrossing
medidata
medio
MedProcure
Melt
metabroadcast
Microsoft
midokura
mnubo
MolloM
monitis
Monty Program
moodyo
MORNINGSTAR
MuleSoft
mydrive
nanigans
napshare
nativeX
NAVTEO
nearbuy
NetSeer
neustar
news360
NEWSWHIP
NEXTAG
ngmoco:)
NREL
nSCALED
nTokio
NUTANIX
NYSE EURONEXT
ONEsite
onespot
onesto
onsip
OOYALA
OSI
OpenAdExchange
OpenMarket
openNMS
opensource connection
opentracker
OPENWAVE
OpenX
opsmatic
orange
Other Inbox
outbrain
PACE
paddypower
page.rss
Palantir
PANTHEON
paper.li
Parse
parseix
PCPhase
PEARSON
PELOTON
pentaho
PeopleBrowsr
PERFECT WORLD
PETER-SERVICE
PING4
PitneyBowes
Pivotal
POLYVORE
PRESS
priceline.com
proofpoint
PROS.
ProtectWise
PULSAR
qafe
Qualtera
quandl
quantcast
quantiFind
QUOVA
Rakuten
rapir
RAZORlabs
ReachLocal
real
Recruiting.com
RED BRICK MEDIA
referentia
RelateIQ
Relay42
Reltio
resulily
retailigence
rethink
Rhapsody
RIGHTSCALE
riptide io
RockMelt
rockyou media
Ruckus
simbase
Sabre
SAFEWAY
SanDisk
SBR
SBRnet
SCALE UNLIMITED
Scandit
Scoop.it!
screen9
SeaChange
SecondSync
SEEMOTO
session m
SEVEN
sevocity
ShareThis
SHAZAM
SHIFT
SHODAN
shopsavvy
shore.li
Simple Energy
SimpleGeo
simplereach
singlewire
SIQUENCE
sirca
SITA
sitevision
СКБ КОНМУР
skillpages
sky
smeet
SMX
SocialArtisan
SOCIALFLOW
SOCIALPARENT
Software Projects
Sonico
Sony Entertainment Network
SoundCloud
SourceNinja
spidertracks
splunk>
Spotify
spotright
spotXchange
Spredfast
Springer
springfuse
springpad
SPROUTSOCIAL
Startapp
Stipple
Stormpath
stratio
Strmr
struq
sumologic
SUNCORP
SwiftKey
Symantec
SYNAPSE
Synchronoss
Taboola
talentica
talkbits
Teads
teamfocus
Telefónica
Tellybug
TENDRIL
echonest
THEFIND
LINKBUILDERS
The New York Times
The Weather Channel
thinglink.
THOMSON REUTERS
TILDE
tink
TinyCo
trapit
travelocity
Twiil
TRICAST
trivadis
trueffect
Tutao
Twenga
TWIMPACT
twitpic
ULTRAVISUAL
UNBXD
V2i
Vast
VEEZIO
verticalscope
ViaSat
videoplaza
VigLink
VisualDNA
VONJOUR
VTEX
W2O
wajam
Walmart
wayin
Webcollage
where
WIDESPACE
wiggle
WILDFIRE
WILLIAMS-SONOMA
COMMERCE
WORKWARESYSTEMS
WSO2
Wuna
Xcerra
Xebia
xobni
YONDEUEO
zangbezang
zeebox
Zemanta
zimp
zipwhip
ZIPZAPPLAY
zoemob
ZOHO
ZONAR
Zonefox
zoominfo
ZOOSK
zoralabs
zscaler

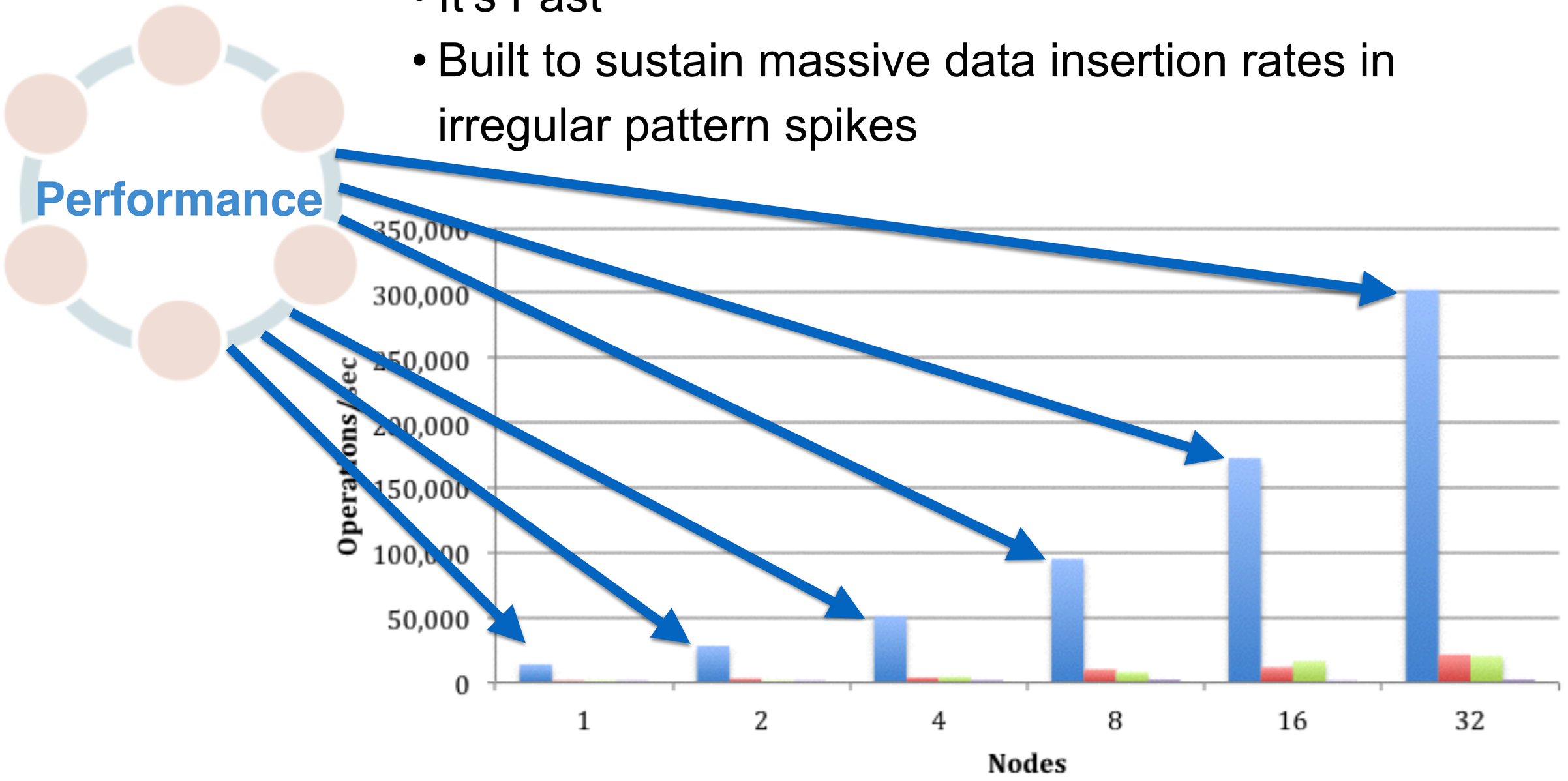
Apache Cassandra



- Scales Linearly to as many nodes as you need
- Scales whenever you need

Apache Cassandra

- It's Fast
- Built to sustain massive data insertion rates in irregular pattern spikes



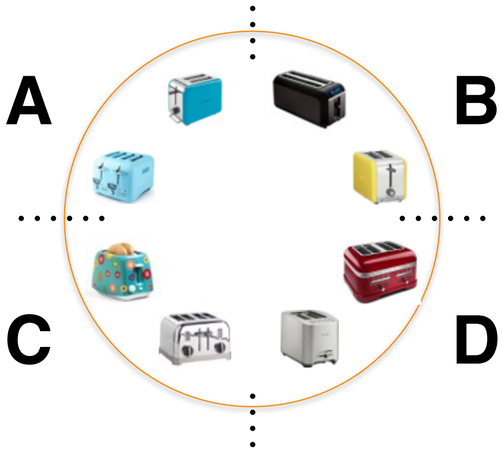
Apache Cassandra

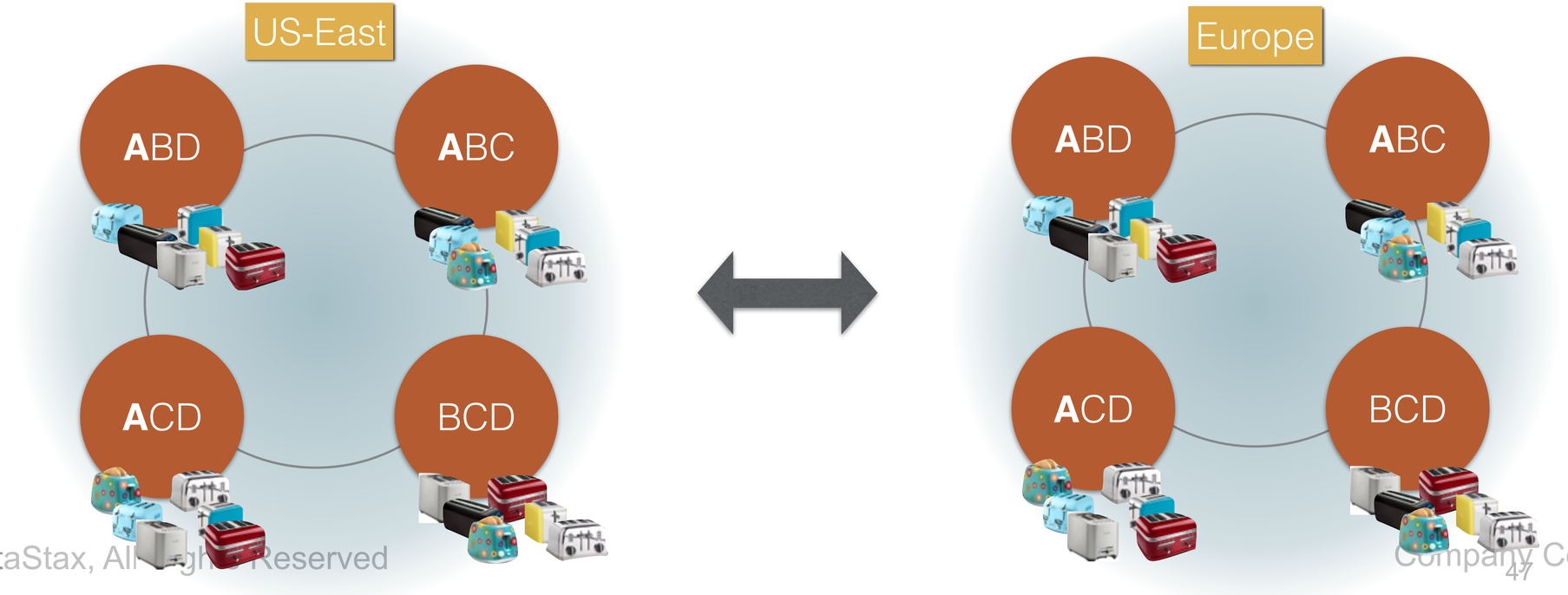
- Automatic Replication
- Multi Datacenter
- Decentralized - no single point of failure
- Survive regional outages
- New nodes automatically add themselves to the cluster
- DataStax drivers automatically discover new nodes

Fault Tolerance & Replication

How many copies of a data should exist in the cluster?

ReplicationFactor=3



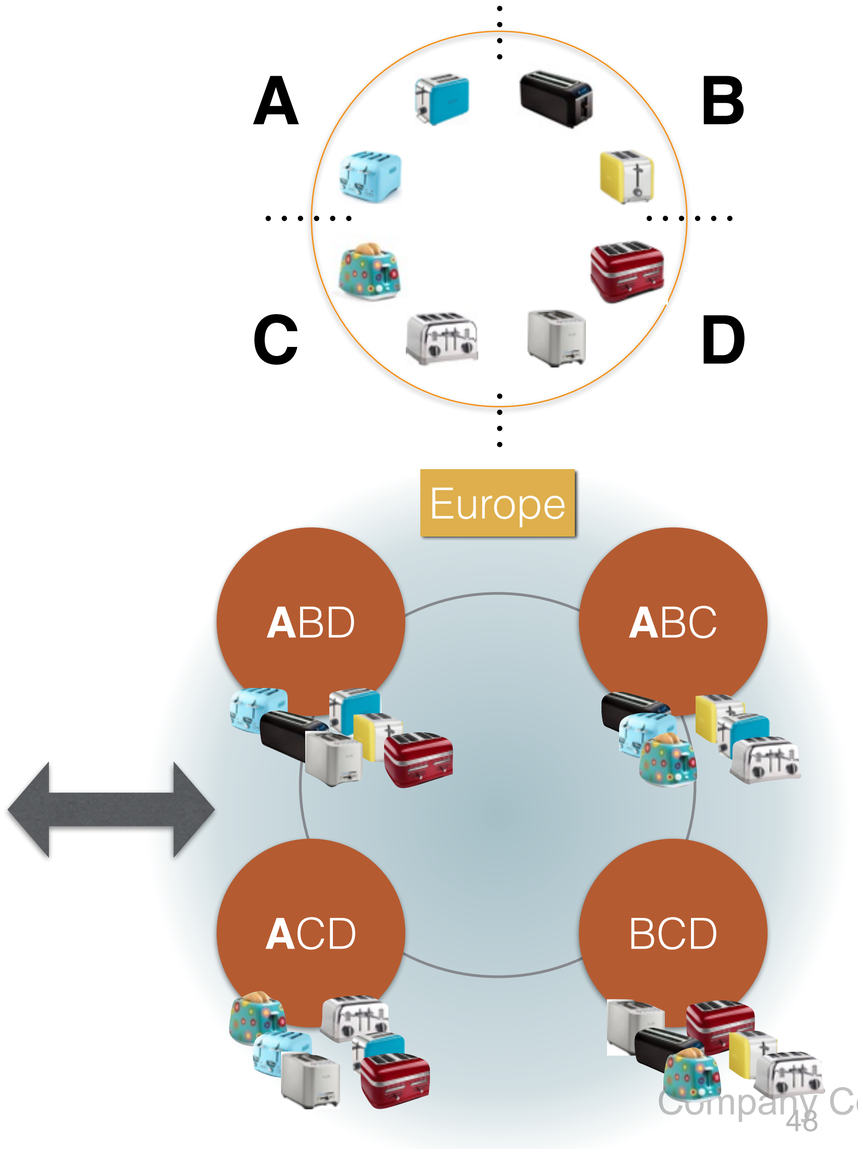


Fault Tolerance & Replication

DATASTAX

How many copies of a data should exist in the cluster?

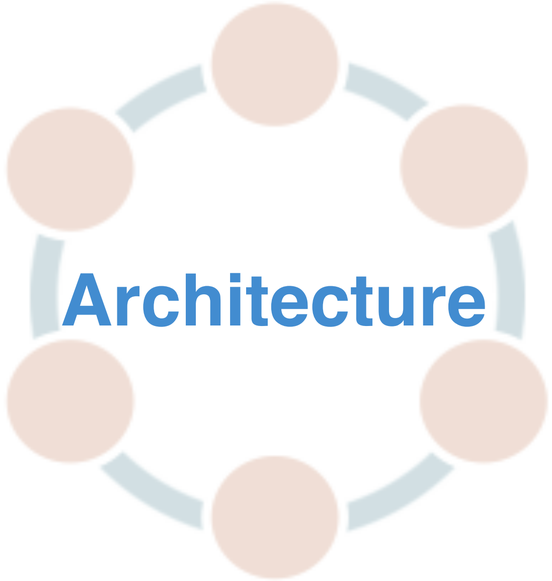
ReplicationFactor=3



Apache Cassandra

- Consensus - Paxos Protocol
- Sequential Read / Write - Timeseries
- Tunable Consistency
- Gossip:

Apache Cassandra



- Distributed, Masterless Ring Architecture
- Network Topology Aware
- Flexible, Schemaless - your data structure can evolve seamlessly over time

Cassandra @ Netflix

- 90% of streaming data is stored in Cassandra
- Data ranges from customer details to Viewing history to streaming bookmarks
- High availability, Multi-region resiliency and Active-Active

C* At CERN: Large Haldron Colider

- ATLAS - Largest of several detectors along the Large Hadron Collider
- Measures particle production when protons collide at a very high center of mass energy

- - Bursty traffic
- - Volume of data from sensors requires
  - - Very large trigger and data acquisition system
  - - 30,000 applications on 2,000 nodes

Genetics / Biological Computations


REACTIVE DATA SECURITY
PREDICTIVE DATA SECURITY
CARTOONSTOCK .com
Search ID: jcen1193

IoT


I THINK MY NEST SMOKE ALARM IS GOING OFF. GOOGLE ADWORDS JUST PITCHED ME A FIRE EXTINGUISHER AND AN OFFER FOR TEMPORARY HOUSING.
TOM FISH BURNE
©marketoonist.com

CQL - Easy

DATASTAX

```
CREATE TABLE users (
   username varchar,
   firstname varchar,
   lastname varchar,
   email list<varchar>,
   password varchar,
   created_date timestamp,
   PRIMARY KEY (username)
);
```

- Familiar syntax
- Many Tools & Drivers
- Many Languages
- Friendly to programmers
- Paxos for locking

```
INSERT INTO users (username, firstname, lastname,
 email, password, created_date)
VALUES ('hedelson','Helena','Edelson',
['helena.edelson@datastax.com'],'ba27e03fd95e507daf2937c937d499ab','2014-11-15 13:50:00')
IF NOT EXISTS;
```

Timeseries Data

DATASTAX

```
CREATE TABLE weather.raw_data (
    wsid text, year int, month int, day int, hour int,
    temperature double, dewpoint double, pressure double,
    wind_direction int, wind_speed double, one_hour_precip
    PRIMARY KEY ((wsid), year, month, day, hour)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC, hour DESC);
```

C* Clustering Columns

Writes by most recent
Reads return most recent first

Cassandra will automatically sort by most recent for both write and read

A record of every event, in order in which it happened, per URL:

```
CREATE TABLE IF NOT EXISTS requests_ks.timeline (
  timesegment bigint, url text, t_uuid timeuuid, method text, headers map <text, text>, body text,
  PRIMARY KEY ((url, timesegment) , t_uuid)
);
```

timeuuid protects from simultaneous events over-writing one another.

timesegment protects from writing unbounded partitions.

```
val multipleStreams = (1 to numDstreams).map { i =>
  streamingContext.receiverStream[HttpRequest](new HttpReceiver(port))
}

streamingContext.union(multipleStreams)
                .map { httpRequest => TimelineRequestEvent(httpRequest)}
                .saveToCassandra("requests_ks", "timeline")
```

Spark Cassandra Connector

Spark Cassandra Connector

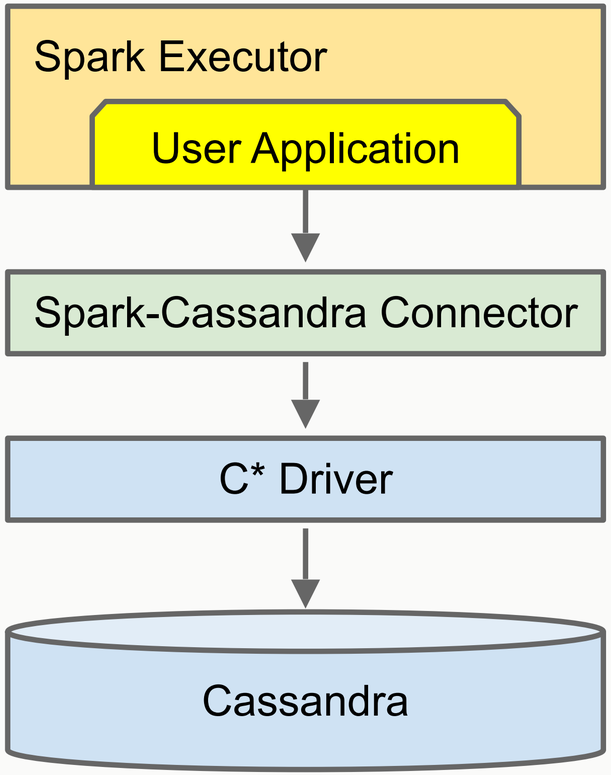
DATASTAX

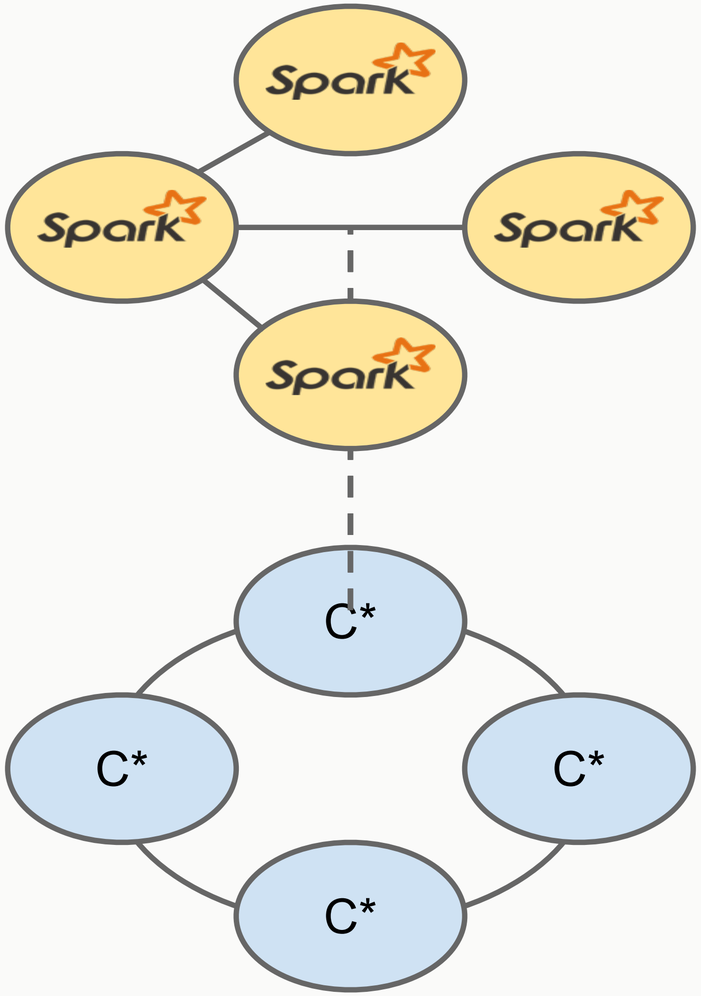
https://github.com/datastax/spark-cassandra-connector

- NOSQL JOINS!
- Write & Read data between Spark and Cassandra
- Compatible with Spark 1.3
- Handles Data Locality for Speed
- Implicit type conversions
- Server-Side Filtering - SELECT, WHERE, etc.
- Natural Timeseries Integration

Spark Cassandra Connector

DATASTAX





Writing and Reading

DATASTAX

SparkContext

```
import com.datastax.spark.connector._
```

StreamingContext

```
import com.datastax.spark.connector.streaming._
```

Write from Spark to Cassandra

DATASTAX

SparkContext

Keyspace

Table

```
sc.parallelize(Seq(0,1,2)).saveToCassandra(“keyspace”, "raw_data")
```

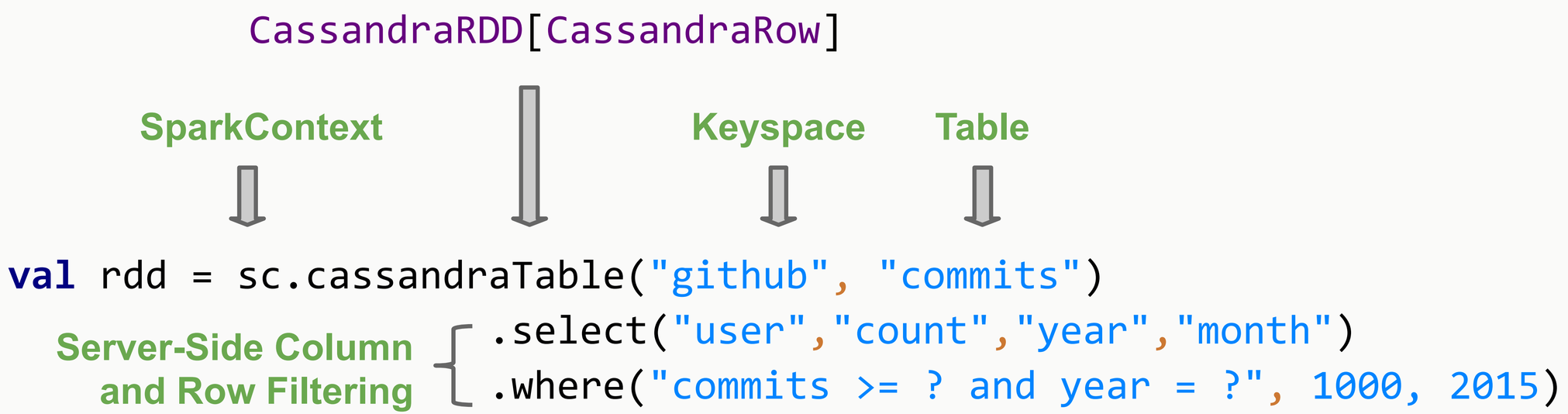
Spark RDD

JOIN with NOSQL!

```
predictionsRdd.join(music).saveToCassandra("music", "predictions")
```

Read From C* to Spark

DATASTAX



Rows: Custom Objects

DATASTAX

StreamingContext CassandraRow Keyspace Table




```
val rdd = ssc.cassandraTable[MonthlyCommits]("github", "commits_aggregate")
              .where("user = ? and project_name = ? and year = ?",
                 "helena", "spark-cassandra-connector", 2015)
```

Rows

DATASTAX

```
val tuplesRdd = sc.cassandraTable[(Int,Date,String)](db, tweetsTable)
 .select("cluster_id","time", "cluster_name")
 .where("time > ? and time < ?",
        "2014-07-12 20:00:01", "2014-07-12 20:00:03”)
```

```
val rdd = ssc.cassandraTable[MyDataType]("stats", "clustering_time")
             .where("key = 1").limit(10).collect
```

```
val rdd = ssc.cassandraTable[(Int,DateTime,String)]("stats", "clustering_time")
             .where("key = 1").withDescOrder.collect
```

Cassandra User Defined Types

DATASTAX

```
CREATE TYPE address (
  street text,
  city text,
  zip_code int,
  country text,
  cross_streets set<text>
);
```

UDT = Your Custom Field Type In Cassandra

Cassandra UDT's With JSON

DATASTAX

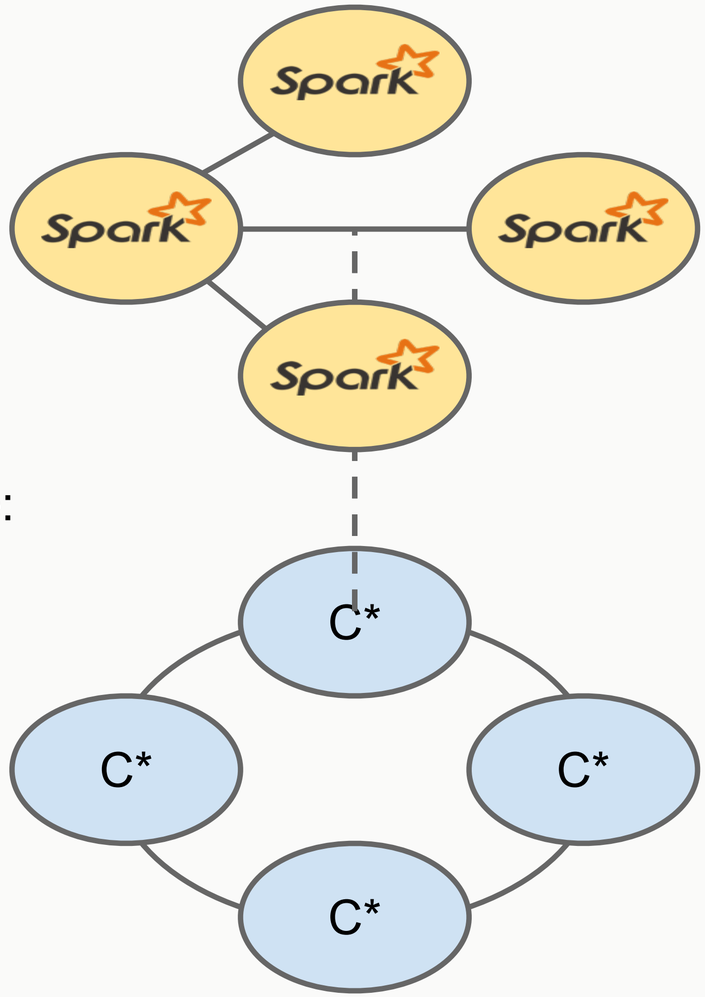
```
{
   "productId": 2,
   "name": "Kitchen Table",
   "price": 249.99,
   "description" : "Rectangular table with oak finish",
   "dimensions": {
      "units": "inches",
      "length": 50.0,
      "width": 66.0,
      "height": 32
   },
   "categories": {
      {
         "category" : "Home Furnishings" {
            "catalogPage": 45,
            "url": "/home/furnishings"
         },
         {
         "category" : "Kitchen Furnishings" {
            "catalogPage": 108,
            "url": "/kitchen/furnishings"
      }
   }
}
```

```
CREATE TYPE dimensions (
   units text,
   length float,
   width float,
   height float
);


CREATE TYPE category (
   catalogPage int,
   url text
);

CREATE TABLE product (
   productId int,
   name text,
   price float,
   description text,
   dimensions frozen <dimensions>,
   categories map <text, frozen <category>>,
   PRIMARY KEY (productId)
);
```

Data Locality

DATASTAX

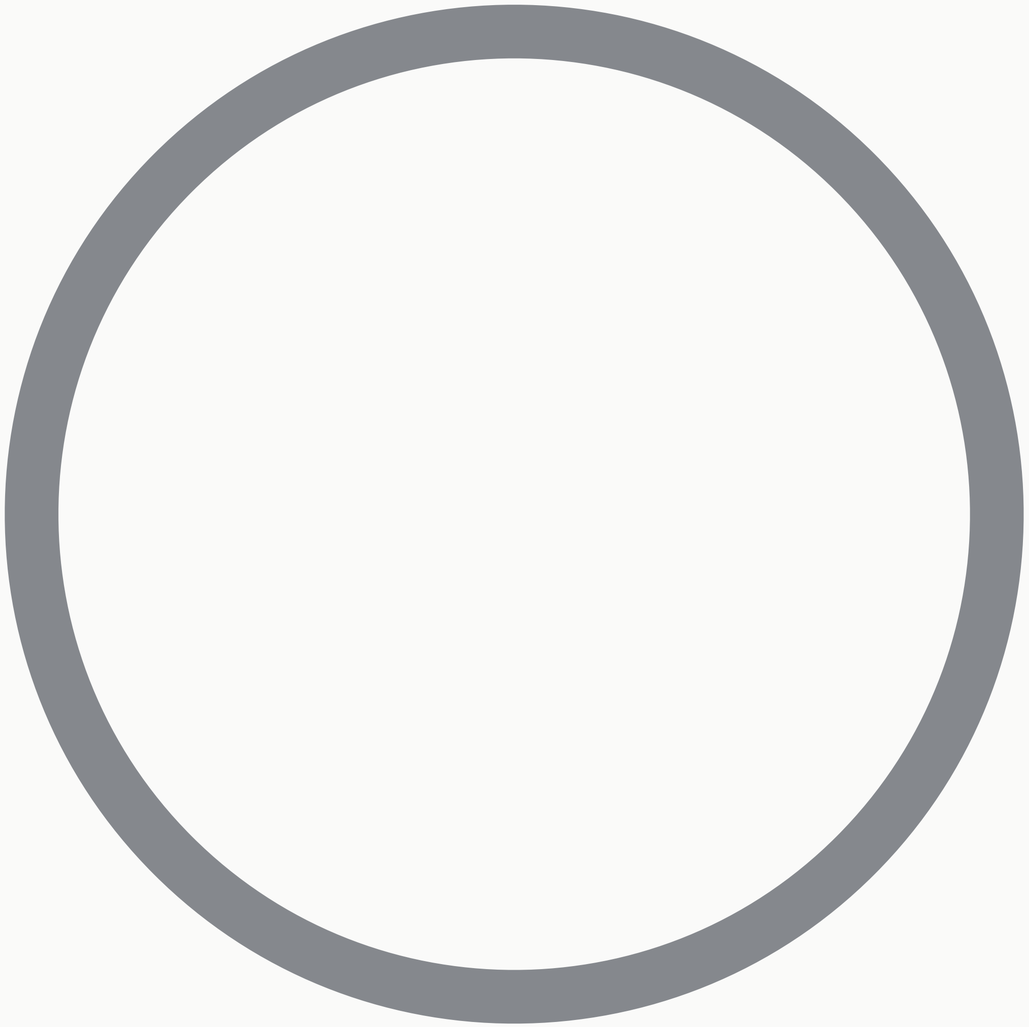
- Spark asks an RDD for a list of its partitions (splits)
- Each split consists of one or more token-ranges
- For every partition
  - Spark asks RDD for a list of preferred nodes to process on
  - Spark creates a task and sends it to one of the nodes for execution



Every Spark task uses a CQL-like query to fetch data for the given token range:

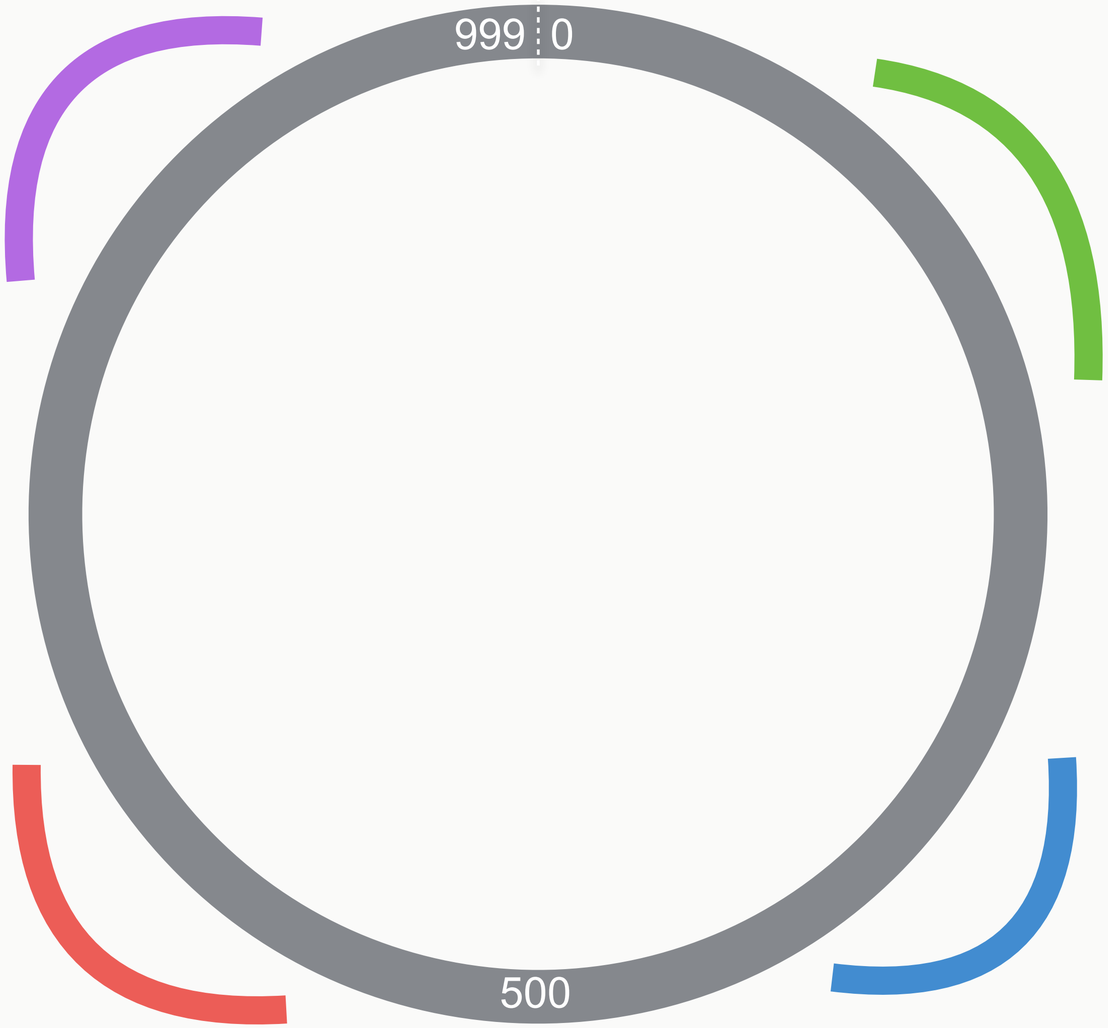
```
SELECT "key", "value"
FROM "test"."kv"
WHERE
  token("key") >  59559742092113932 1 AND
  token("key") <= 59559743119420013 2
ALLOW FILTERING
```

Cassandra Locates a Row Based on Partition Key and Token Range



All of the rows in a Cassandra Cluster are stored based based on their location in the *Token Range*.

Cassandra Locates a Row Based on Partition Key and Token Range



New York City/ Manhattan: Helena

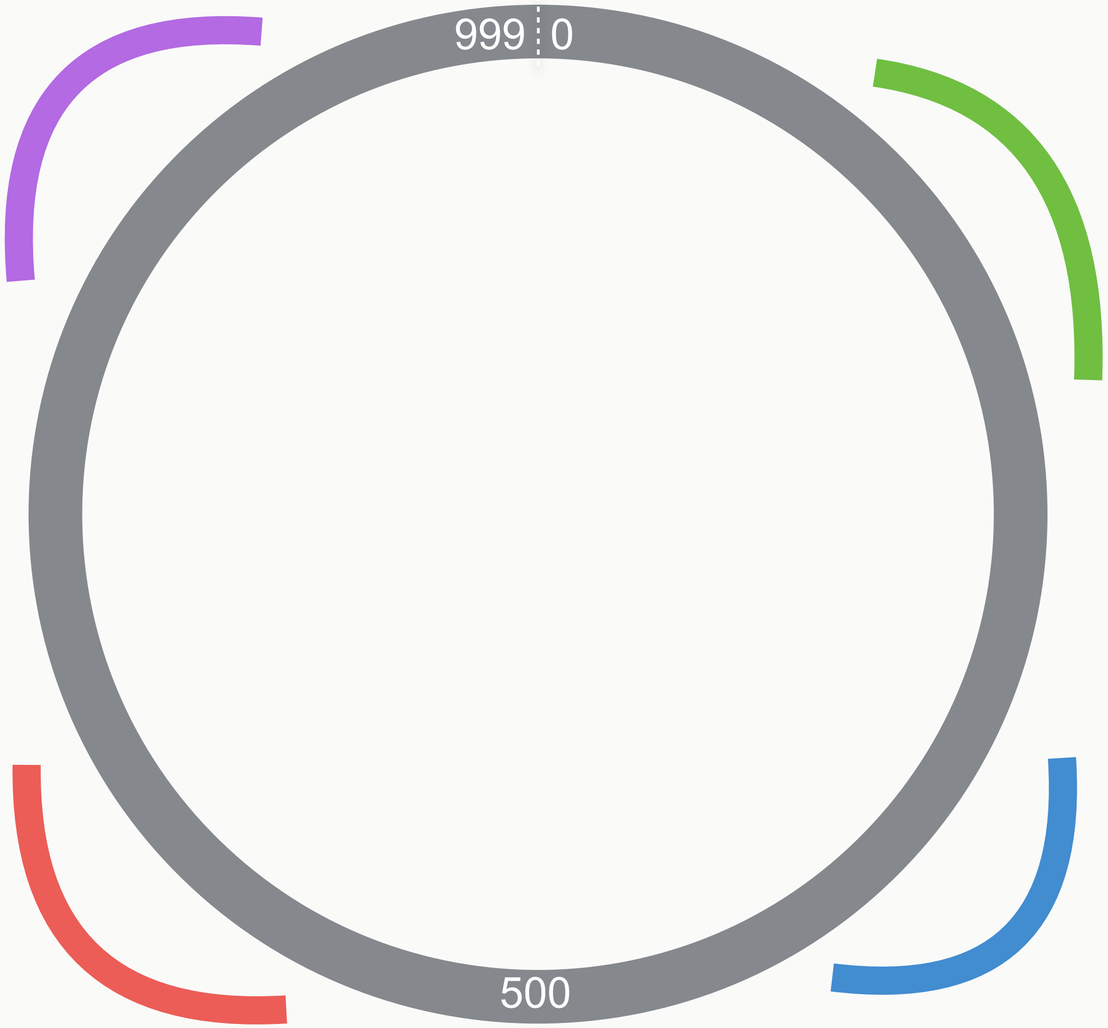
Warsaw: Piotr & Jacek

St. Petersburg: Artem

San Francisco: Brian,Russell & Alex

Each of the *Nodes* in a Cassandra Cluster is primarily responsible for one set of *Tokens.*

Cassandra Locates a Row Based on Partition Key and Token Range



New York City

750 - 99

Warsaw

350 - 749

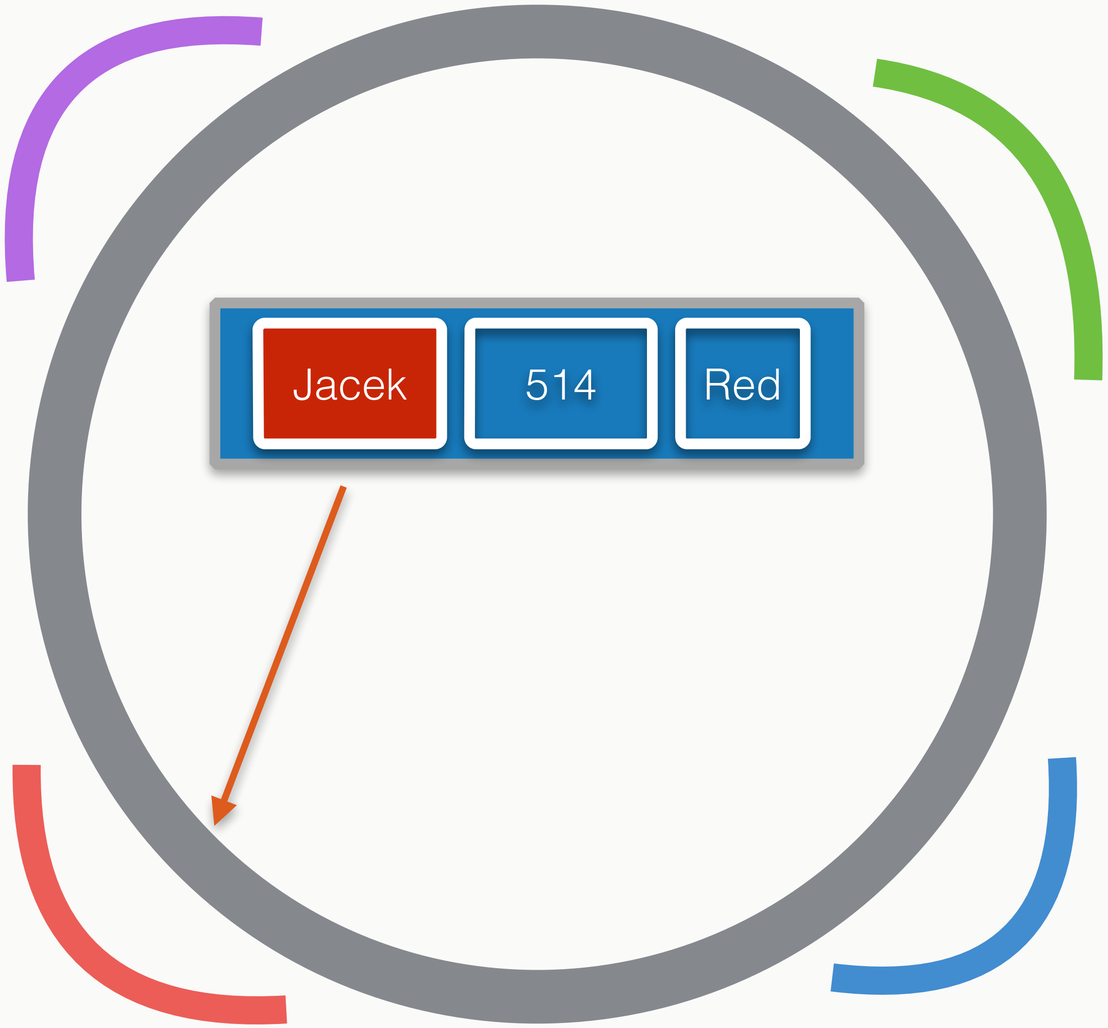
St. Petersburg

San Francisco

100 - 349

Each of the *Nodes* in a Cassandra Cluster is primarily responsible for one set of *Tokens.*

Cassandra Locates a Row Based on Partition Key and Token Range



New York City

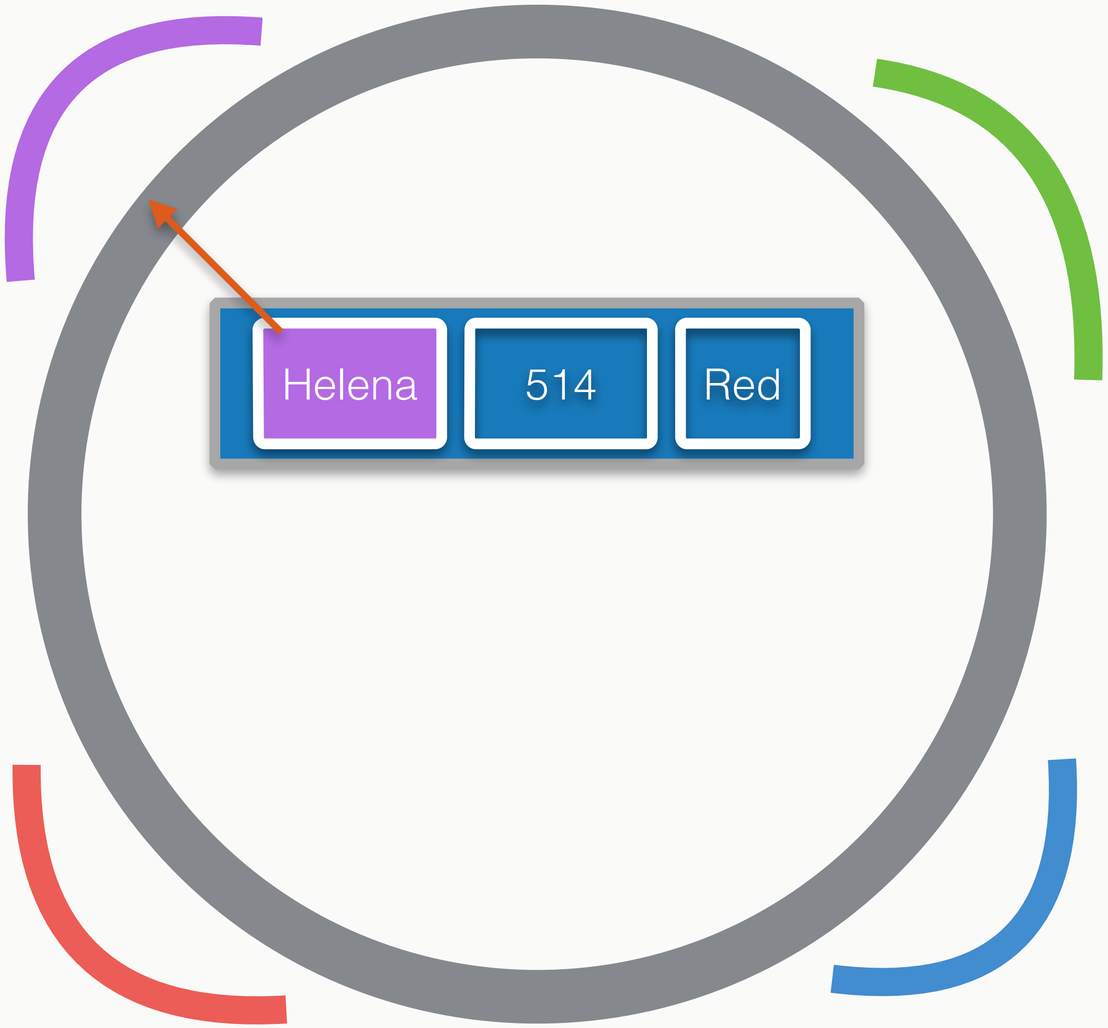
Warsaw

St. Petersburg

San Francisco

The CQL Schema designates at least one column to be the *Partition Key.*

Cassandra Locates a Row Based on Partition Key and Token Range



New York City

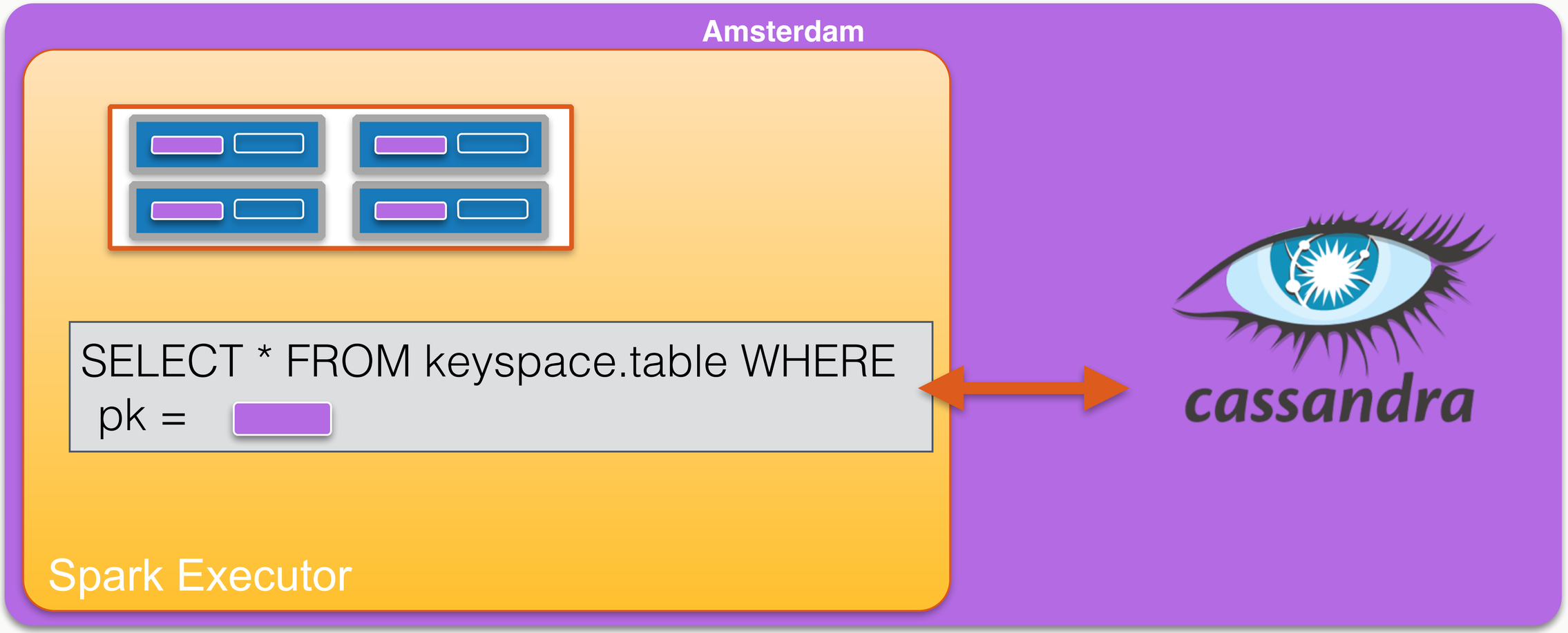
Warsaw

St. Petersburg

San Francisco

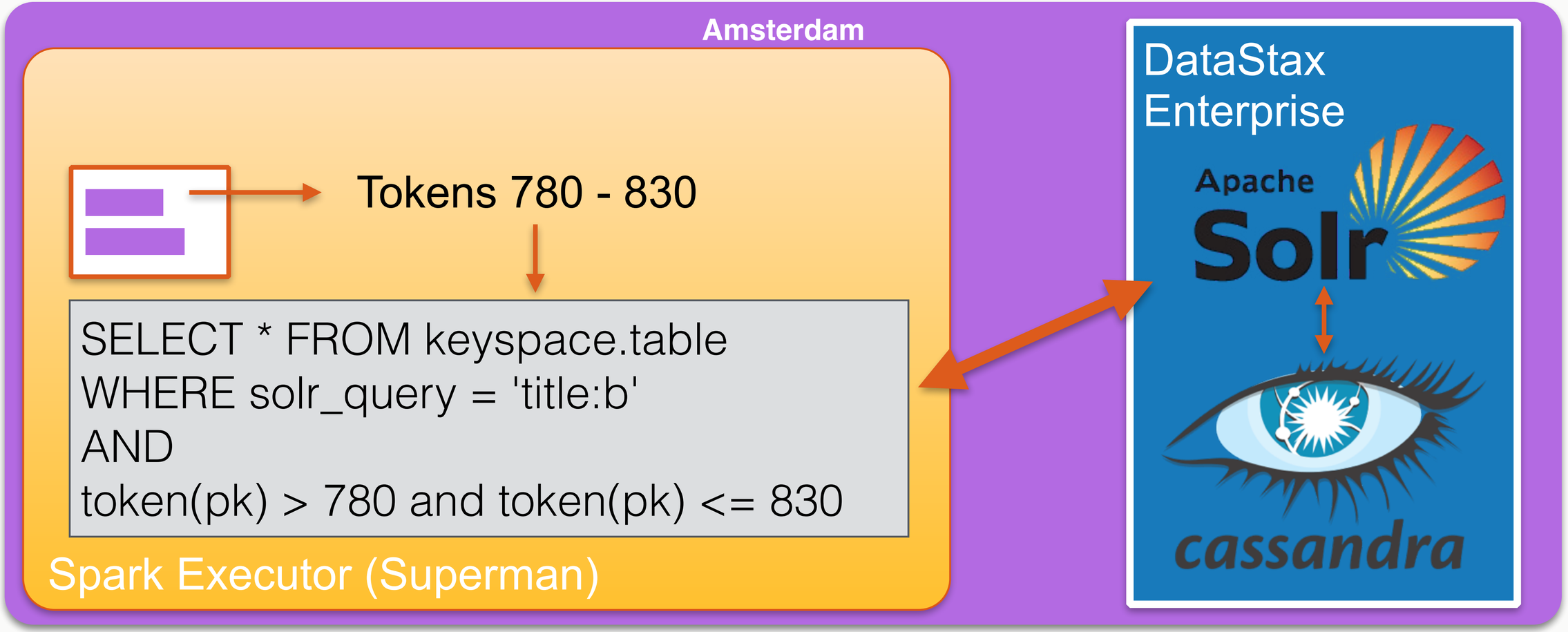
The hash of the *Partition Key* tells us where a row should be stored.

The Spark Executor uses the Connector to Pull Rows from the Local Cassandra Instance



The C* Driver pages `spark.cassandra.input.page.row.size` CQL rows at a time

DataStax Enterprise Enables This Same Machinery with Solr Pushdown



Composable Pipelines
With Spark, Kafka & Cassandra

Spark SQL with Cassandra

DATASTAX

```
import org.apache.spark.sql.cassandra.CassandraSQLContext

val cc = new CassandraSQLContext(sparkContext)
cc.setKeyspace(keyspaceName)
cc.sql("""
  SELECT table1.a, table1.b, table.c, table2.a
  FROM table1 AS table1
  JOIN table2 AS table2 ON table1.a = table2.a
  AND table1.b = table2.b
  AND table1.c = table2.c
  """)
.map(Data(_))
.saveToCassandra(keyspace1, table3)
```

Spark SQL with Cassandra & JSON

DATASTAX

```
cqlsh> CREATE TABLE github_stats.commits_aggr(user VARCHAR PRIMARY KEY, commits INT…);
```

```
val sql = new SQLContext(sparkContext)

val json = Seq(
  """{"user":"helena","commits":98, "month":3, "year":2015}""",
  """{"user":"jacek-lewandowski", "commits":72, "month":3, "year":2015}""",
  """{"user":"pkolaczk", "commits":42, "month":3, "year":2015}""")

// write
sql.jsonRDD(json)
   .map(CommitStats(_))
   .flatMap(compute)
   .saveToCassandra("stats","monthly_commits")

// read
val rdd = sc.cassandraTable[MonthlyCommits]("stats","monthly_commits")
```

Spark Streaming, Kafka, C* and JSON

```
KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
  ssc, kafkaParams, topicMap, StorageLevel.MEMORY_ONLY)
  .map { case (_,json) => JsonParser.parse(json).extract[MonthlyCommits]}
  .saveToCassandra("github_stats","commits_aggr")
```

```
cqlsh> select * from github_stats.commits_aggr;

 user              | commits | month | year
-------------------+---------+-------+------
          pkolaczk |      42 |     3 | 2015
 jacek-lewandowski |      43 |     3 | 2015
            helena |      98 |     3 | 2015
(3 rows)
```

Kafka Streaming Word Count

DATASTAX

```
sparkConf.set("spark.cassandra.connection.host", "10.20.3.45")
val streamingContext = new StreamingContext(conf, Seconds(30))


KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
 streamingContext, kafkaParams, topicMap, StorageLevel.MEMORY_ONLY)
  .map(_._2)
  .countByValue()
  .saveToCassandra("my_keyspace","wordcount")
```

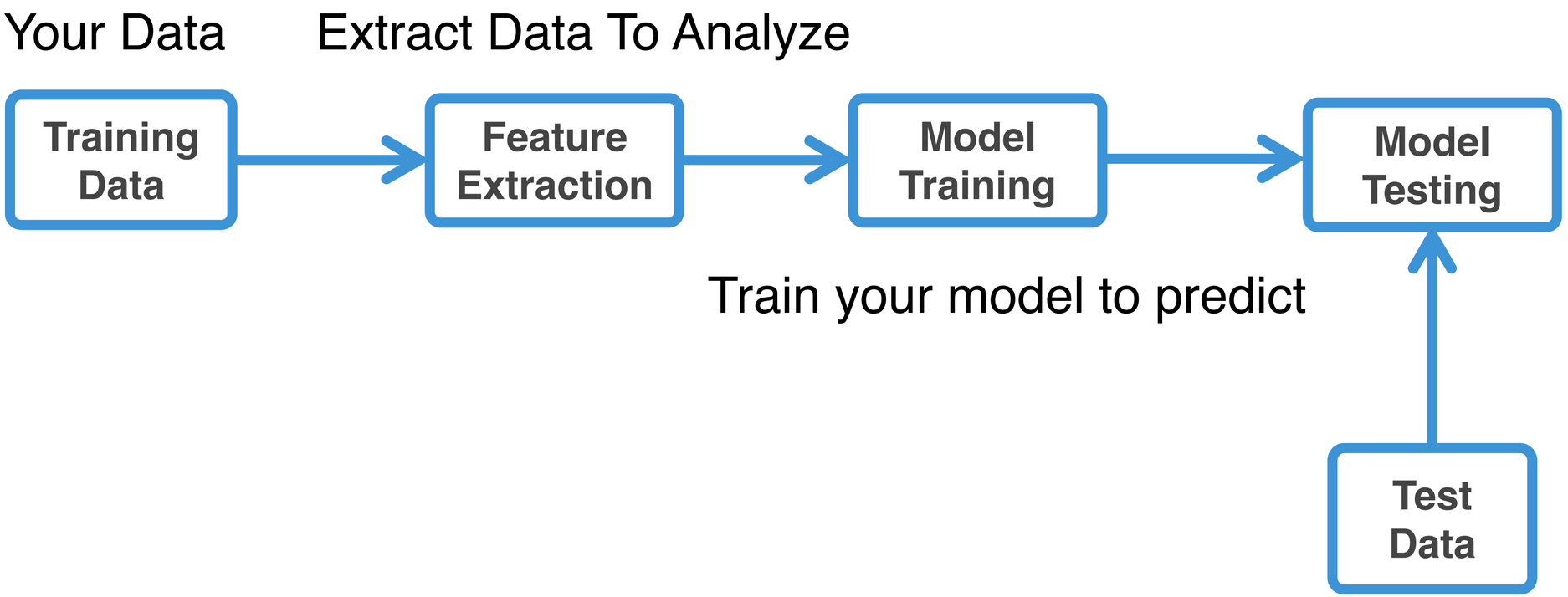
Spark Streaming, Twitter & Cassandra

DATASTAX

```
CREATE TABLE IF NOT EXISTS keyspace.table (
  topic text, interval text, mentions counter,
  PRIMARY KEY(topic, interval)
) WITH CLUSTERING ORDER BY (interval DESC)
```

```
/** Cassandra is doing the sorting for you here. */
TwitterUtils.createStream(
    ssc, auth, tags, StorageLevel.MEMORY_ONLY_SER_2)
  .flatMap(_.getText.toLowerCase.split("""\s+"""))
  .filter(tags.contains(_))
  .countByValueAndWindow(Seconds(5), Seconds(5))
  .transform((rdd, time) =>
      rdd.map { case (term, count) => (term, count, now(time))})
  .saveToCassandra(keyspace, table)
```

Spark MLLib


Your Data
Extract Data To Analyze
Training Data
Feature Extraction
Model Training
Model Testing
Train your model to predict
Test Data

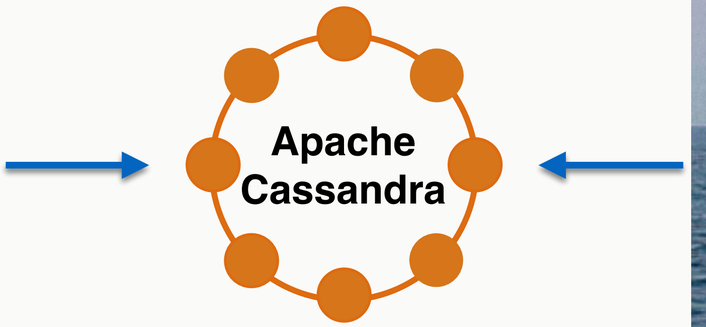
Spark Streaming ML, Kafka & C*

DATASTAX

```
val ssc = new StreamingContext(new SparkConf()…, Seconds(5)

val testData = ssc.cassandraTable[String](keyspace,table).map(LabeledPoint.parse)

val trainingStream = KafkaUtils.createStream[K, V, KDecoder, VDecoder](
                      ssc, kafkaParams, topicMap, StorageLevel.MEMORY_ONLY)
                     .map(_._2).map(LabeledPoint.parse)

trainingStream.saveToCassandra("ml_keyspace", “raw_training_data")

val model = new StreamingLinearRegressionWithSGD()
  .setInitialWeights(Vectors.dense(weights))
  .trainOn(trainingStream)


//Making predictions on testData
  model
  .predictOnValues(testData.map(lp => (lp.label, lp.features)))
  .saveToCassandra("ml_keyspace", "predictions")
```

KillrWeather

- Global sensors & satellites collect data
- Cassandra stores in sequence
- Application reads in sequence



Data model should look like your queries

Queries I Need

- Get data by ID
- Get data for a single date and time
- Get data for a window of time
- Compute, store and retrieve daily, monthly, annual aggregations

Design Data Model to support queries

- Store raw data per ID
- Store time series data in order: most recent to oldest
- Compute and store aggregate data in the stream
- Set TTLs on historic data

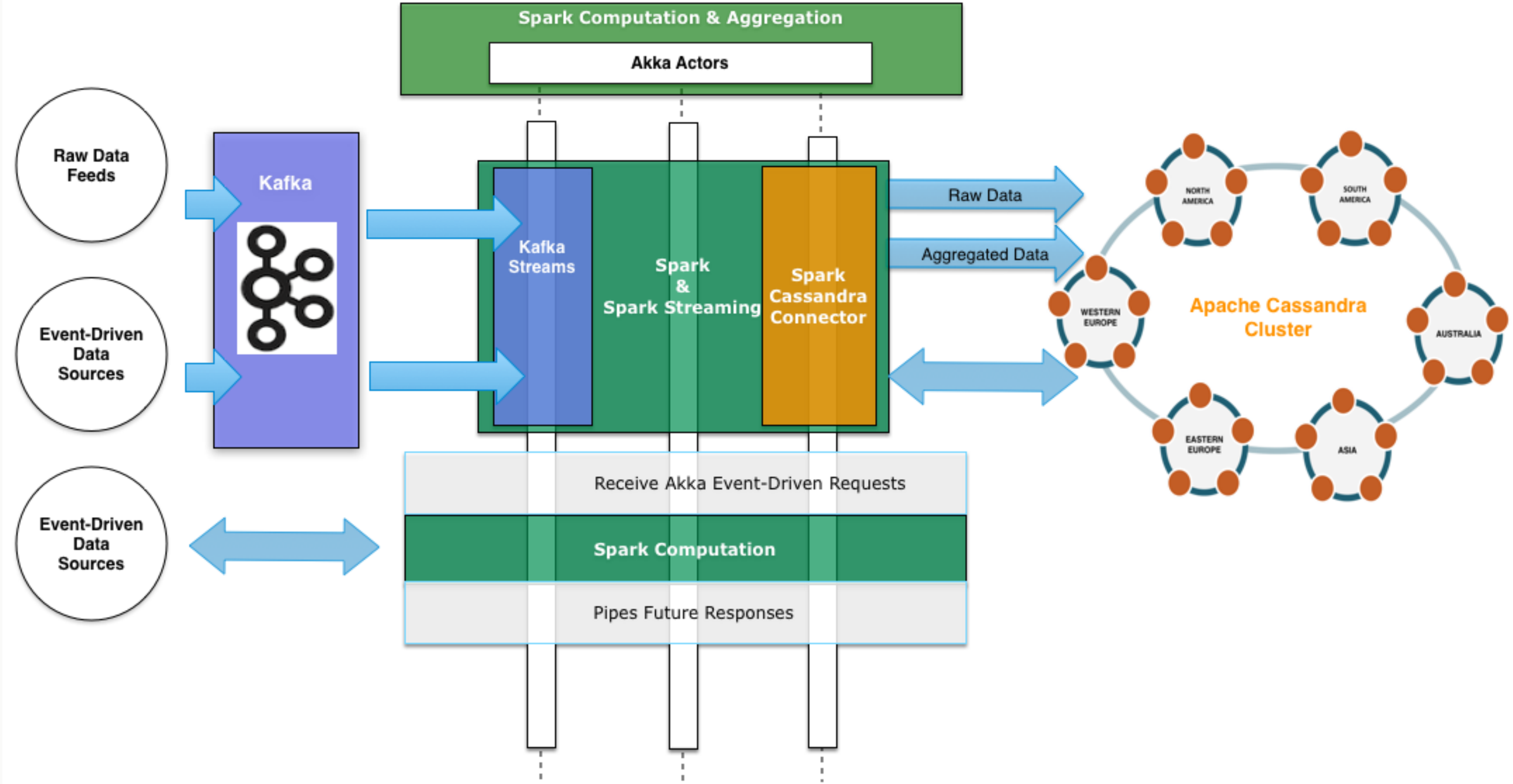
Data Model

DATASTAX

```
CREATE TABLE daily_temperature (
   weather_station text,
   year int,
   month int,
   day int,
   hour int,
   temperature double,
   PRIMARY KEY (weather_station,year,month,day,hour)
);
```

- Weather Station Id and Time are unique
- Store as many as needed

```
INSERT INTO temperature(weather_station,year,month,day,hour,temperature)
VALUES (‘10010:99999’,2005,12,1,7,-5.6);

INSERT INTO temperature(weather_station,year,month,day,hour,temperature)
VALUES (‘10010:99999’,2005,12,1,8,-5.1);

INSERT INTO temperature(weather_station,year,month,day,hour,temperature)
VALUES (‘10010:99999’,2005,12,1,9,-4.9);

INSERT INTO temperature(weather_station,year,month,day,hour,temperature)
VALUES (‘10010:99999’,2005,12,1,10,-5.3);
```


Spark Computation & Aggregation
Akka Actors
Raw Data Feeds
Event-Driven Data Sources
Event-Driven Data Sources
Kafka
Kafka Streams
Spark & Spark Streaming
Spark Cassandra Connector
Raw Data
Aggregated Data
NORTH AMERICA
SOUTH AMERICA
WESTERN EUROPE
Apache Cassandra Cluster
AUSTRALIA
EASTERN EUROPE
ASIA
Receive Akka Event-Driven Requests
Spark Computation
Pipes Future Responses

Load-Balanced Data Ingestion

DATASTAX

```
class HttpNodeGuardian extends ClusterAwareNodeGuardianActor {

  cluster.joinSeedNodes(Vector(..))

  context.actorOf(BalancingPool(PoolSize).props(Props(
      new KafkaPublisherActor(KafkaHosts, KafkaBatchSendSize))))


  Cluster(context.system) registerOnMemberUp {
    context.actorOf(BalancingPool(PoolSize).props(Props(
        new HttpReceiverActor(KafkaHosts, KafkaBatchSendSize))))
  }

  def initialized: Actor.Receive = { … }

}
```

Client: HTTP Receiver Akka Actor

DATASTAX

```
class HttpDataIngestActor(kafka: ActorRef) extends Actor with ActorLogging {
  implicit val system = context.system
  implicit val askTimeout: Timeout = settings.timeout
  implicit val materializer = ActorFlowMaterializer(
    ActorFlowMaterializerSettings(system))

  val requestHandler: HttpRequest => HttpResponse = {
    case HttpRequest(HttpMethods.POST, Uri.Path("/weather/data"), headers, entity, _) =>
        headers.toSource collect { case s: Source =>
          kafka ! KafkaMessageEnvelope[String, String](topic, group, s.data:_*)
        }
        HttpResponse(200, entity = HttpEntity(MediaTypes.`text/html`)
      }.getOrElse(HttpResponse(404, entity = "Unsupported request"))
    case _: HttpRequest =>
      HttpResponse(400, entity = "Unsupported request")
  }

  Http(system).bind(HttpHost, HttpPort).map { case connection  =>
      log.info("Accepted new connection from " + connection.remoteAddress)
      connection.handleWithSyncHandler(requestHandler) }

  def receive : Actor.Receive = {
    case e =>
  }
}
```

Client: Kafka Producer Akka Actor

DATASTAX

```
class KafkaProducerActor[K, V](config: ProducerConfig) extends Actor {

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1.minute) {
      case _: ActorInitializationException   => Stop
      case _: FailedToSendMessageException   => Restart
      case _: ProducerClosedException        => Restart
      case _: NoBrokersForPartitionException => Escalate
      case _: KafkaException                 => Escalate
      case _: Exception                      => Escalate
    }

  private val producer = new KafkaProducer[K, V](producerConfig)

  override def postStop(): Unit = producer.close()

  def receive = {
    case e: KafkaMessageEnvelope[K,V] => producer.send(e)
  }
}
```

DATASTAX

Store raw data on ingestion

Store Raw Data From Kafka Stream To C*

```
val kafkaStream = KafkaUtils.createStream[K, V, KDecoder, VDecoder]
  (ssc, kafkaParams, topicMap, StorageLevel.DISK_ONLY_2)
  .map(transform)
  .map(RawWeatherData(_))

/** Saves the raw data to Cassandra. */
kafkaStream.saveToCassandra(keyspace, raw_ws_data)
```

Now we can replay on failure for later computation, etc

```
/** Now proceed with computations from the same stream.. */
kafkaStream…
```

Let's See Our Data Model Again

DATASTAX

```
CREATE TABLE weather.raw_data (
    wsid text, year int, month int, day int, hour int,
    temperature double, dewpoint double, pressure double,
    wind_direction int, wind_speed double, one_hour_precip
    PRIMARY KEY ((wsid), year, month, day, hour)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC, hour DESC);
```

```
CREATE TABLE daily_aggregate_precip (
    wsid text,
    year int,
    month int,
    day int,
    precipitation counter,
    PRIMARY KEY ((wsid), year, month, day)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC);
```

Efficient Stream Computation

DATASTAX

```
class KafkaStreamingActor(kafkaPm: Map[String, String], ssc: StreamingContext, ws: WeatherSettings)
  extends AggregationActor {
  import settings._

  val kafkaStream = KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
    ssc, kafkaParams, Map(KafkaTopicRaw -> 1), StorageLevel.DISK_ONLY_2)
    .map(_._2.split(","))
    .map(RawWeatherData(_))

  kafkaStream.saveToCassandra(CassandraKeyspace, CassandraTableRaw)

  /** RawWeatherData: wsid, year, month, day, oneHourPrecip */
  kafkaStream.map(hour => (hour.wsid, hour.year, hour.month, hour.day, hour.oneHourPrecip))
  .saveToCassandra(CassandraKeyspace, CassandraTableDailyPrecip)


  /** Now the [[StreamingContext]] can be started. */
  context.parent ! OutputStreamInitialized

  def receive : Actor.Receive = {…}
}
```

Gets the partition key: Data Locality
Spark C* Connector feeds this to Spark

Cassandra Counter column in our schema, no expensive `reduceByKey` needed. Simply let C* do it: not expensive and fast.

```scala
/** For a given weather station, calculates annual cumulative precip - or year to date. */
class PrecipitationActor(ssc: StreamingContext, settings: WeatherSettings) extends AggregationActor {

  def receive : Actor.Receive = {
    case GetPrecipitation(wsid, year)        => cumulative(wsid, year, sender)
    case GetTopKPrecipitation(wsid, year, k) => topK(wsid, year, k, sender)
  }

  /** Computes annual aggregation.Precipitation values are 1 hour deltas from the previous. */
  def cumulative(wsid: String, year: Int, requester: ActorRef): Unit =
    ssc.cassandraTable[Double](keyspace, dailytable)
      .select("precipitation")
      .where("wsid = ? AND year = ?", wsid, year)
      .collectAsync()
      .map(AnnualPrecipitation(_, wsid, year)) pipeTo requester

  /** Returns the 10 highest temps for any station in the `year`. */
  def topK(wsid: String, year: Int, k: Int, requester: ActorRef): Unit = {
    val toTopK = (aggregate: Seq[Double]) => TopKPrecipitation(wsid, year,
      ssc.sparkContext.parallelize(aggregate).top(k).toSeq)

    ssc.cassandraTable[Double](keyspace, dailytable)
      .select("precipitation")
      .where("wsid = ? AND year = ?", wsid, year)
      .collectAsync().map(toTopK) pipeTo requester
  }
}
```

Efficient Batch Analytics

DATASTAX

```
class TemperatureActor(sc: SparkContext, settings: WeatherSettings)
  extends AggregationActor {
  import akka.pattern.pipe

  def receive: Actor.Receive = {
    case e: GetMonthlyHiLowTemperature => highLow(e, sender)
  }

  def highLow(e: GetMonthlyHiLowTemperature, requester: ActorRef): Unit =
    sc.cassandraTable[DailyTemperature](keyspace, daily_temperature_aggr)
      .where("wsid = ? AND year = ? AND month = ?", e.wsid, e.year, e.month)
      .collectAsync()
      .map(MonthlyTemperature(_, e.wsid, e.year, e.month)) pipeTo requester

}
```

C* data is automatically sorted by most recent - due to our data model.

Additional Spark or collection sort not needed.

@helenaedelson

github.com/helena

slideshare.net/helenaedelson

Learn More Online and at Cassandra Summit