Delivering Meaning In Near-Real Time At High Velocity & Massive Scale

Helena Edelson
@helenaedelson

Who Is This Person?

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

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

cassandra

Talk Roadmap

What Delivering Meaning

Why Spark, Kafka, Cassandra & Akka

How Deployment Architecture & Code

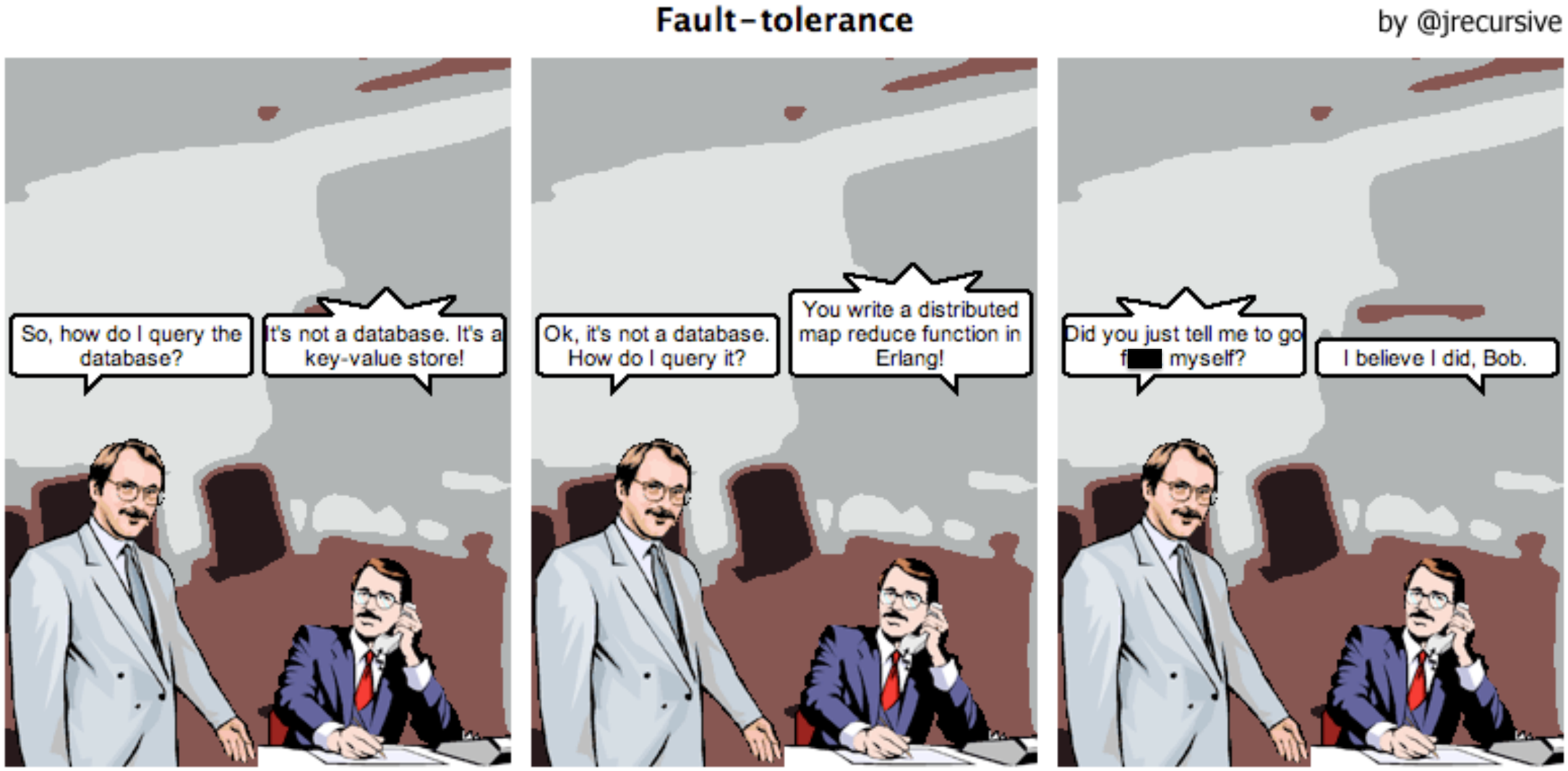
App Robust Implementation

No Time For Questions but…

@helenaedelson

github.com/helena

slideshare.net/helenaedelson



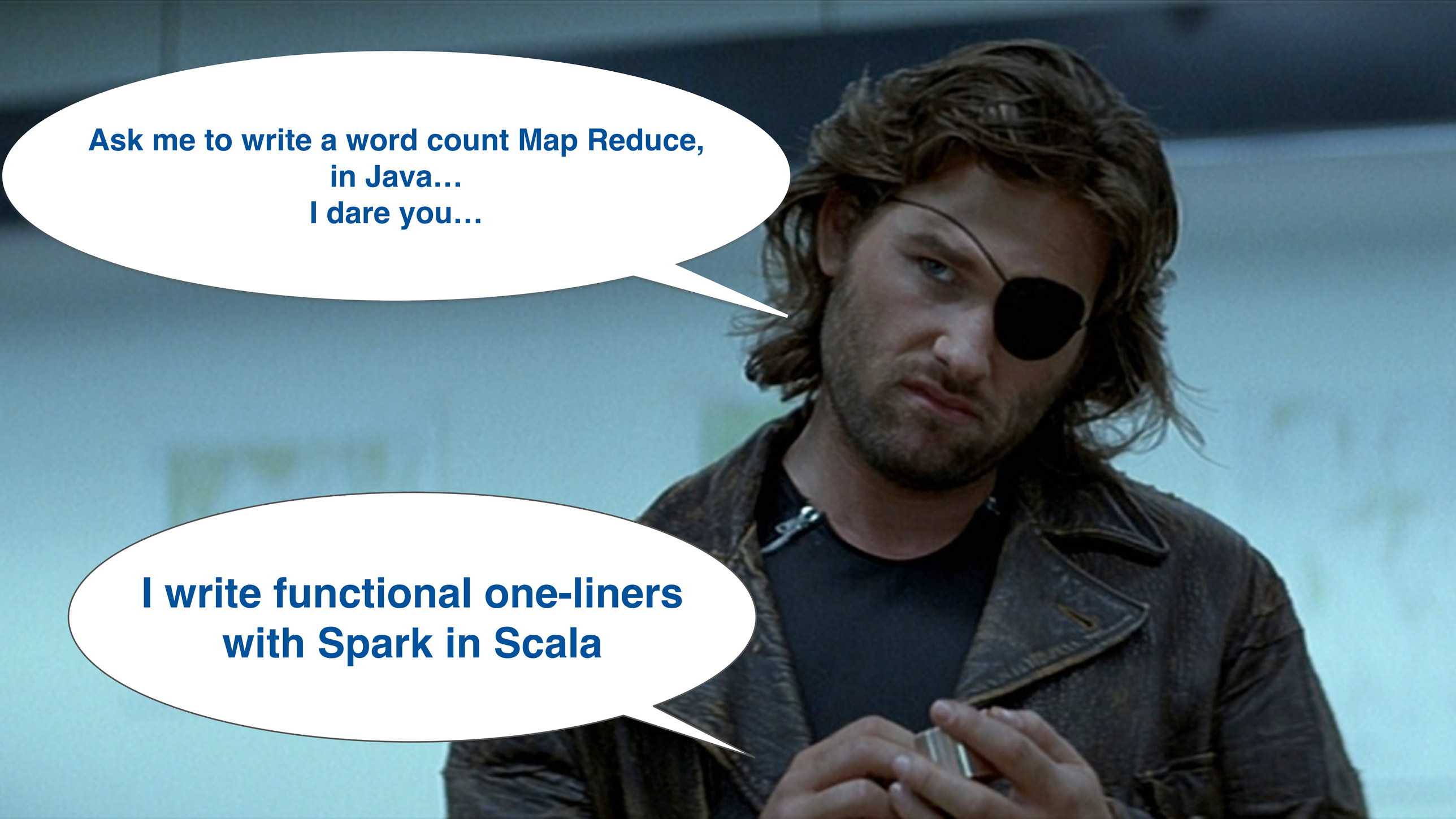
Strategies

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

Strategy	Technologies
Scalable Infrastructure / Elastic scale on demand	Spark, Cassandra, Kafka
Partition For Scale, Network Topology Aware	Cassandra, Spark, Kafka, Akka Cluster
Replicate For Resiliency span racks and datacenters, survive regional outages	Spark,Cassandra, Akka Cluster all hash the node ring
Share Nothing, Masterless decentralized cluster membership	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	Cassandra & Akka Cluster use Paxos Algo & Gossip throughout the node ring
Parallelism	Spark, Cassandra, Kafka, Akka
Asynchronous Message Passing	Kafka, Akka, Spark
Fast, Low Latency, Data Locality	Cassandra, Spark, Kafka
Location Transparency	Akka, Spark, Cassandra, Kafka

Other technologies like Mesos are very helpful too but that's part of a larger discussion.


ESCAPE FROM HADOOP
SPIGOT


Ask me to write a word count Map Reduce,
in Java…
I dare you…
I write functional one-liners
with Spark in Scala

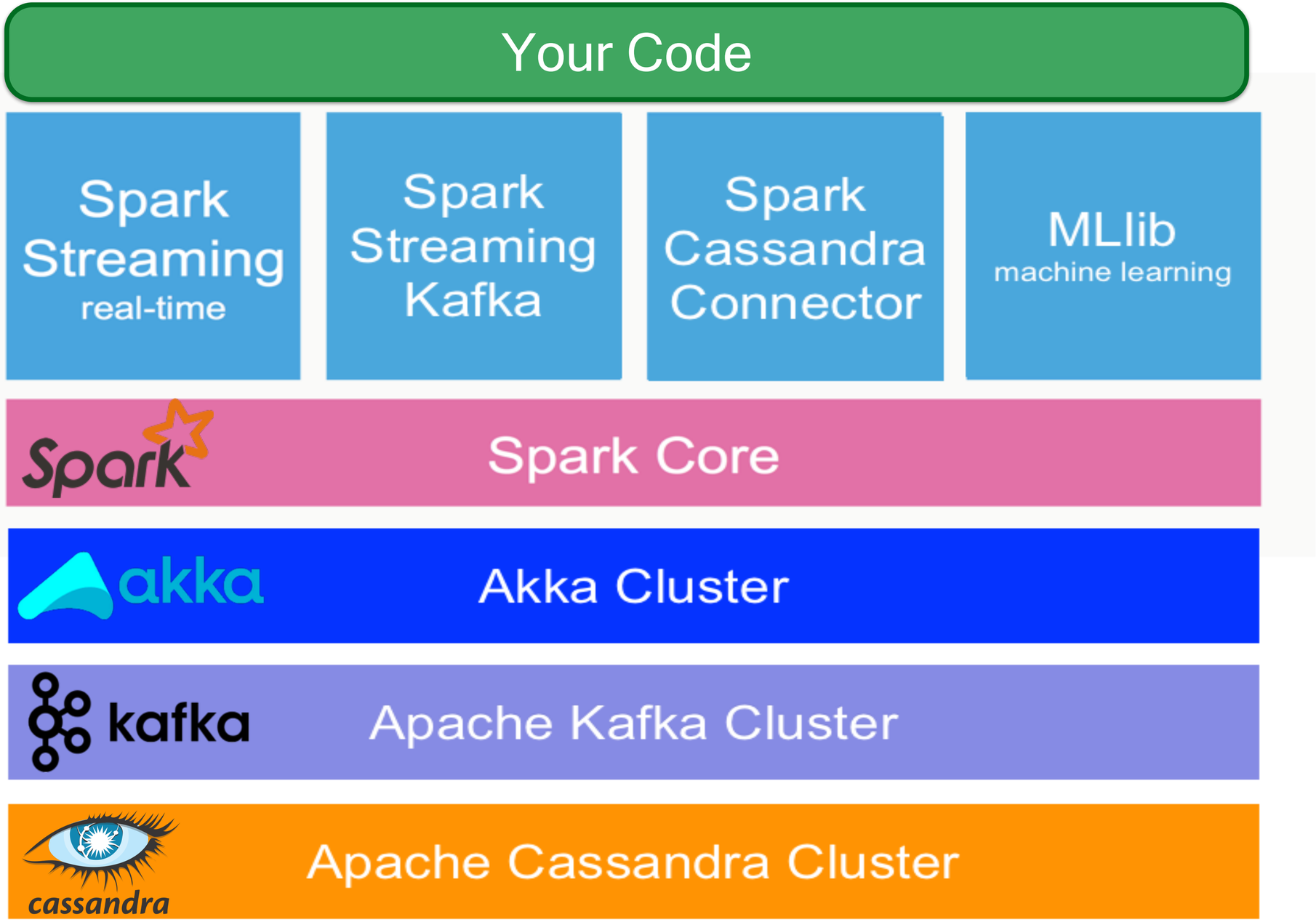
Lambda Architecture

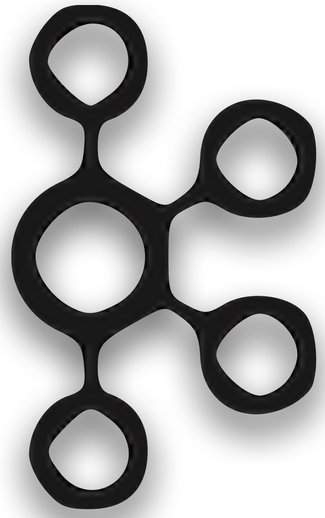
A data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream processing methods.

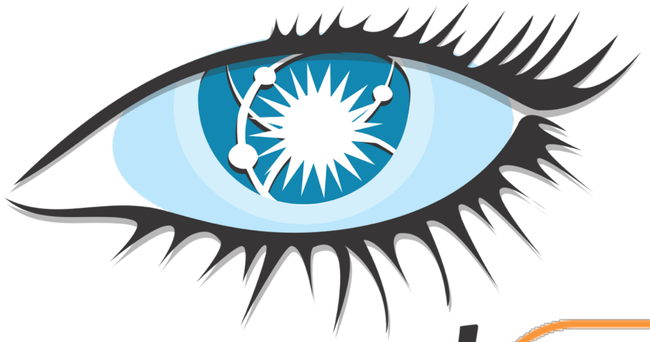
Spark - one of the few data processing framework that allows you to seamlessly integrate batch and stream processing of petabytes of data in the same application.

I need fast access to historical data on the fly for predictive modeling

I also need to be able to replay my data from any point of failure

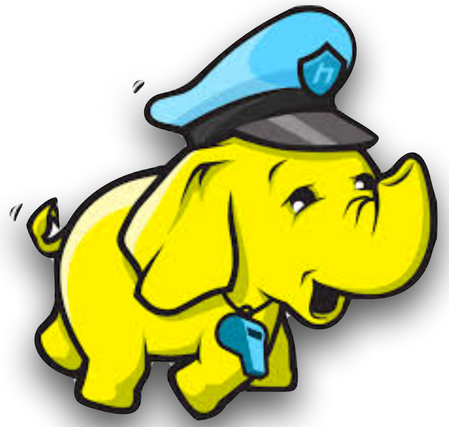

Your Code
Spark Streaming
real-time
Spark Streaming Kafka
Spark Cassandra Connector
MLlib
machine learning
Spark
Spark Core
akka
Akka Cluster
kafka
Apache Kafka Cluster
cassandra
Apache Cassandra Cluster





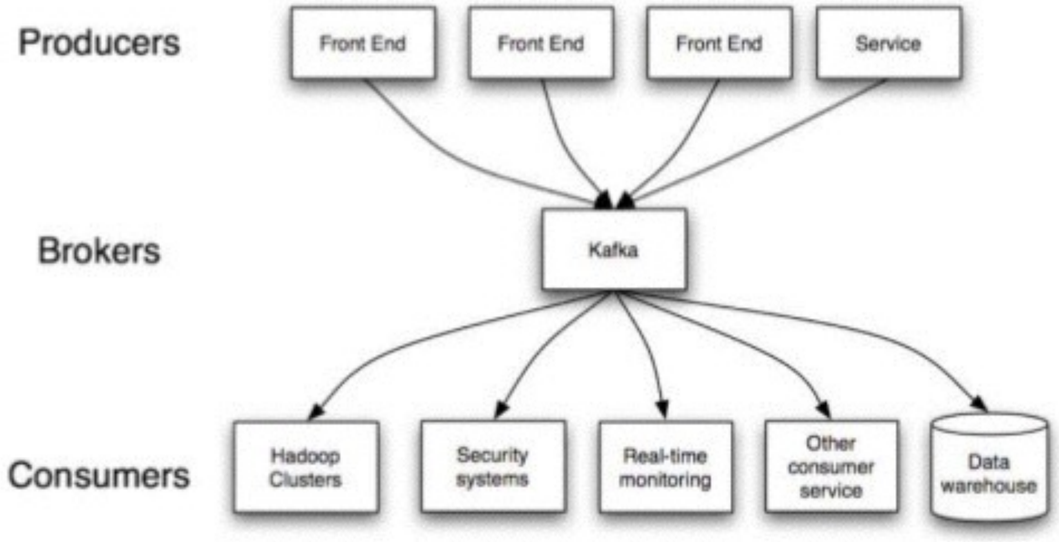
cassandra

Spark Streaming



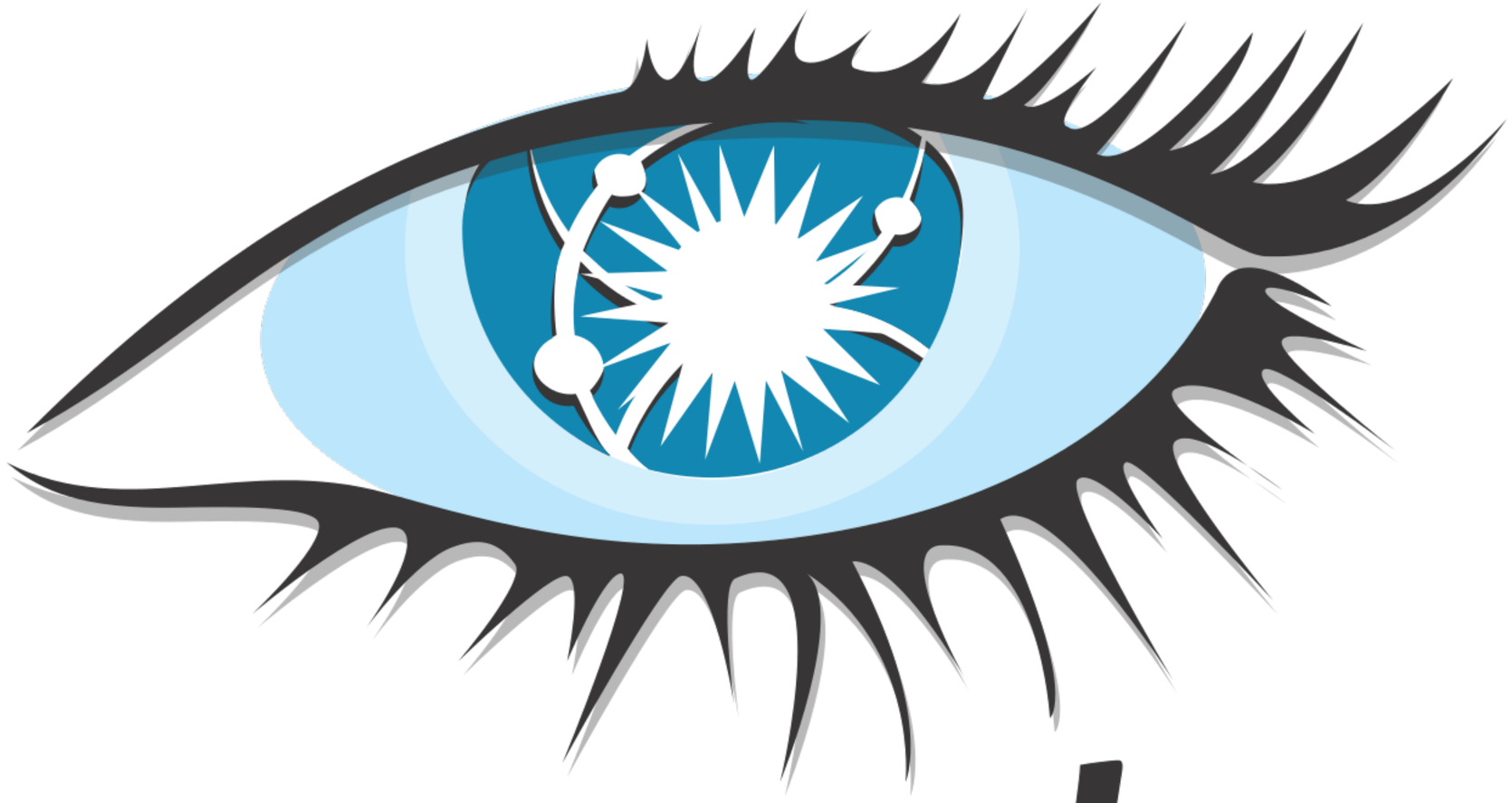
- Massively Scalable
- High Performance
- Always On
- Masterless

Kafka decouples data-pipelines



akka

- Fault tolerant
  - Hierarchical Supervision
  - Customizable Failure Strategies & Detection
- Asynchronous Data Passing
  - Parallelized
  - Adaptive / Predictive
  - Load-Balanced



cassandra

Public Cassandra Users

Spark SUMMIT EAST

Apache Cassandra

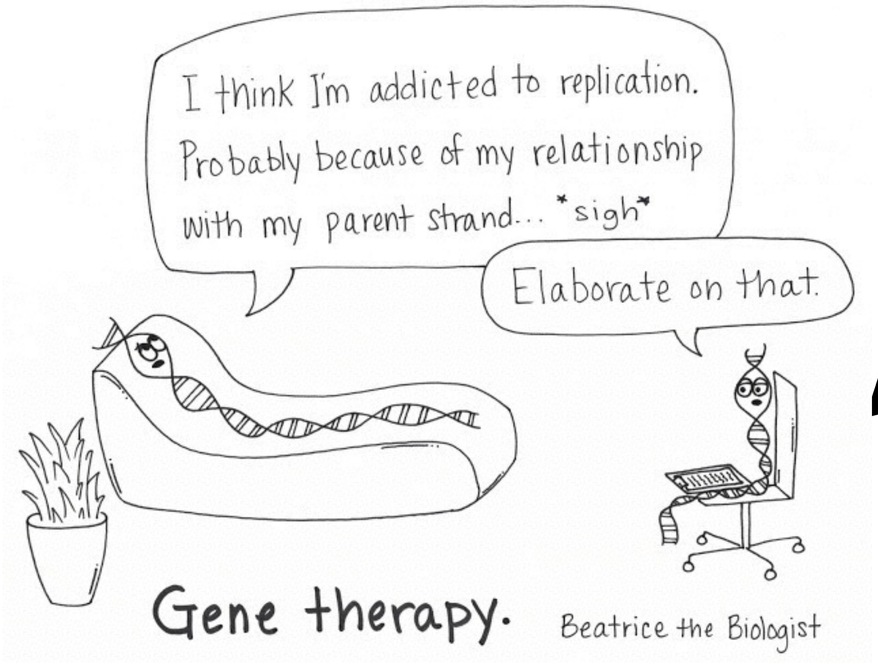
- Elasticity - scale to as many nodes as you need, when you need
- Always On - No single point of failure, Continuous availability
  - Masterless peer to peer architecture
- Replication Across DataCenters
  - Flexible Data Storage
  - Read and write to any node syncs across the cluster
  - Operational simplicity - all nodes in a cluster are the same
- Fast Linear-Scale Performance
- Transaction Support

Always On - No Single Point of Failure

Financial

Security

Scientific Data

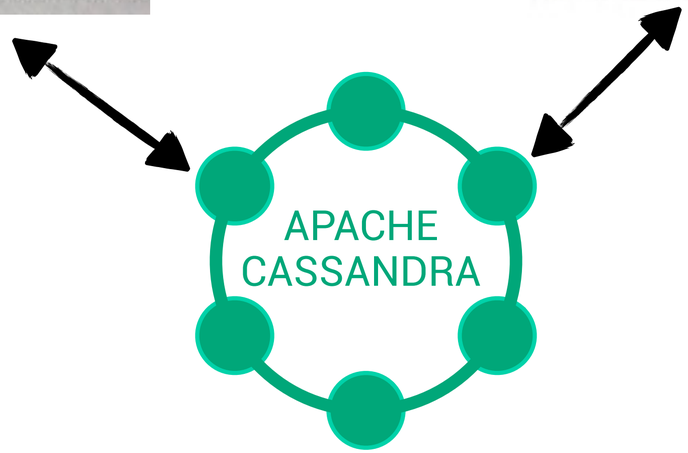


APACHE CASSANDRA

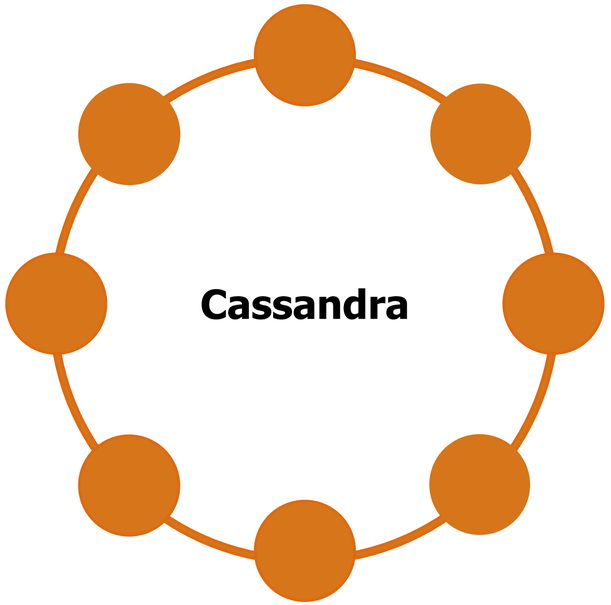
Sensor Data

Data Storage for Machine Learning

IoT



Cassandra



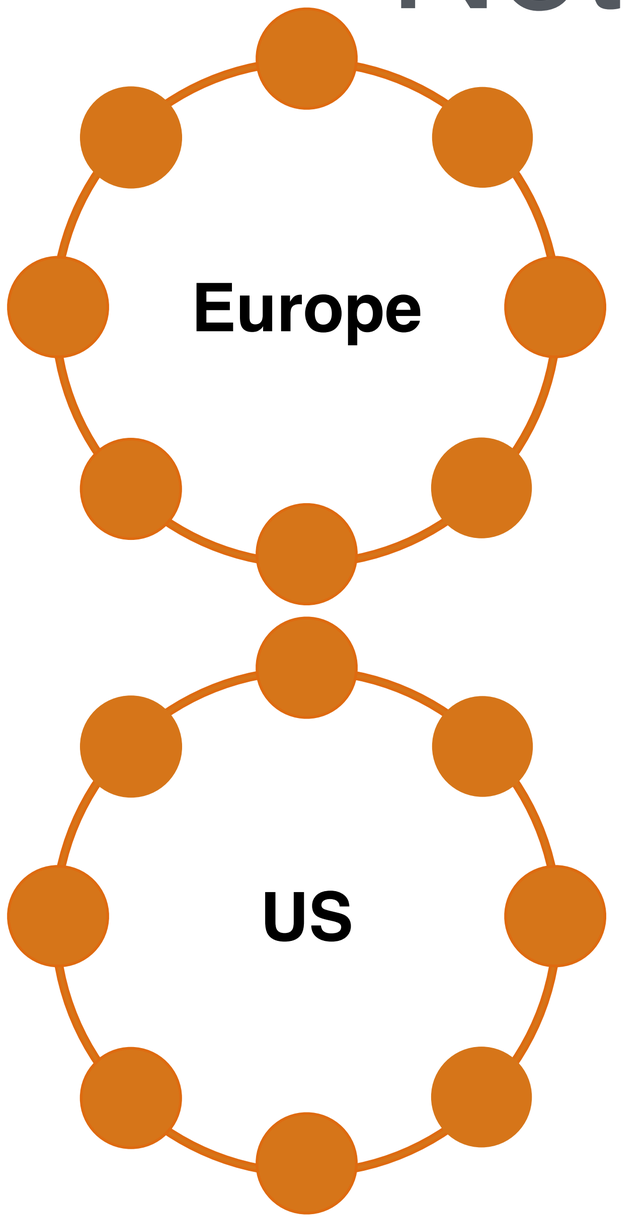
Availability Model

- Amazon Dynamo Paper
- Distributed masterless

Data Model

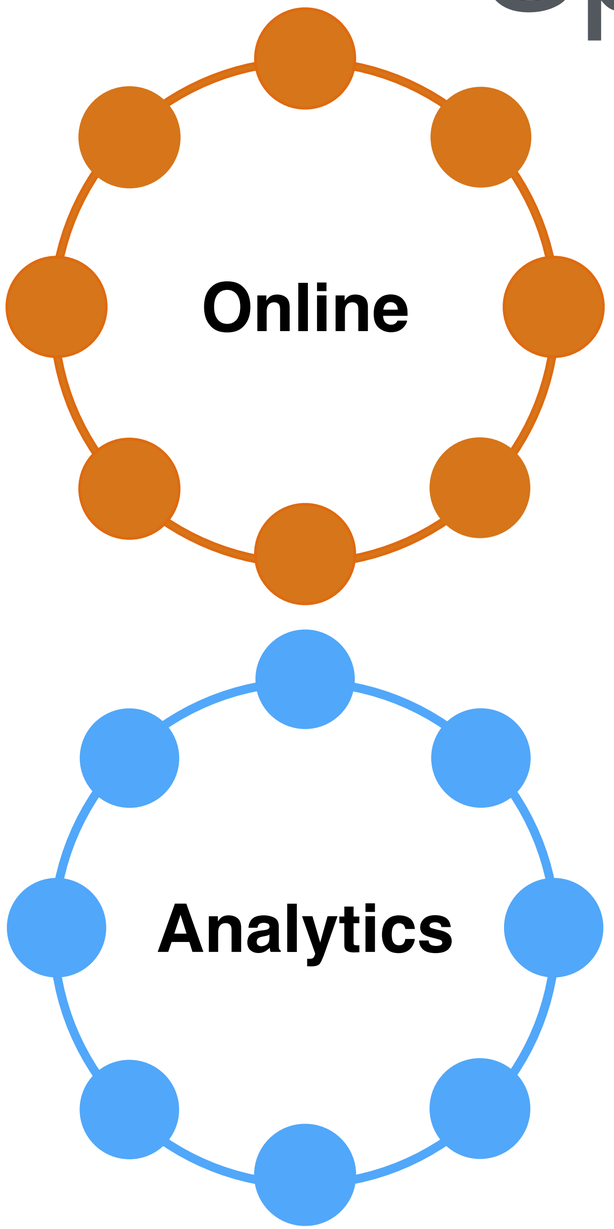
- Google BigTable
- Column family data model

Network Topology Aware



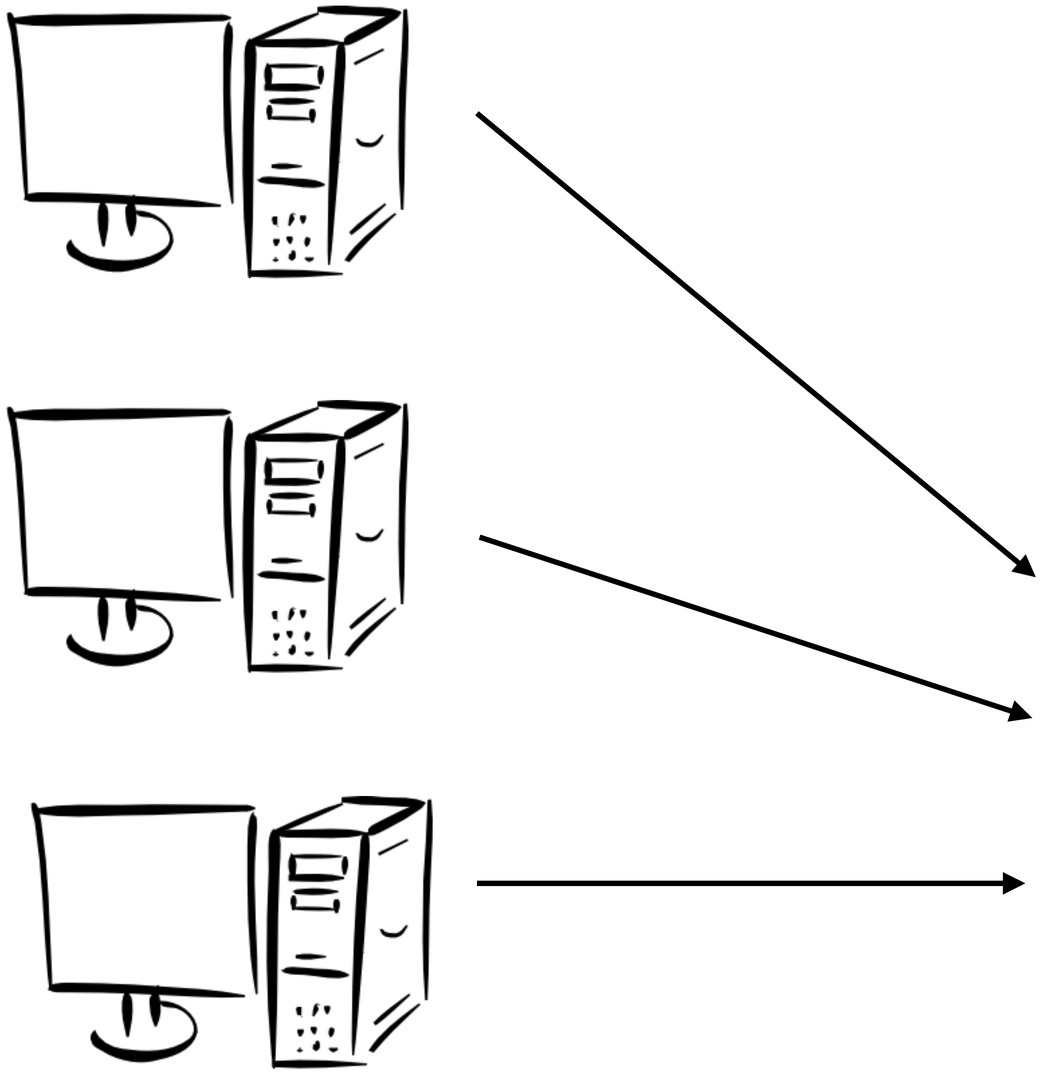
- Availability Model
  - Amazon Dynamo Paper
  - Distributed masterless
- Data Model
  - Google BigTable
  - Column family data model
- Multi data center replication

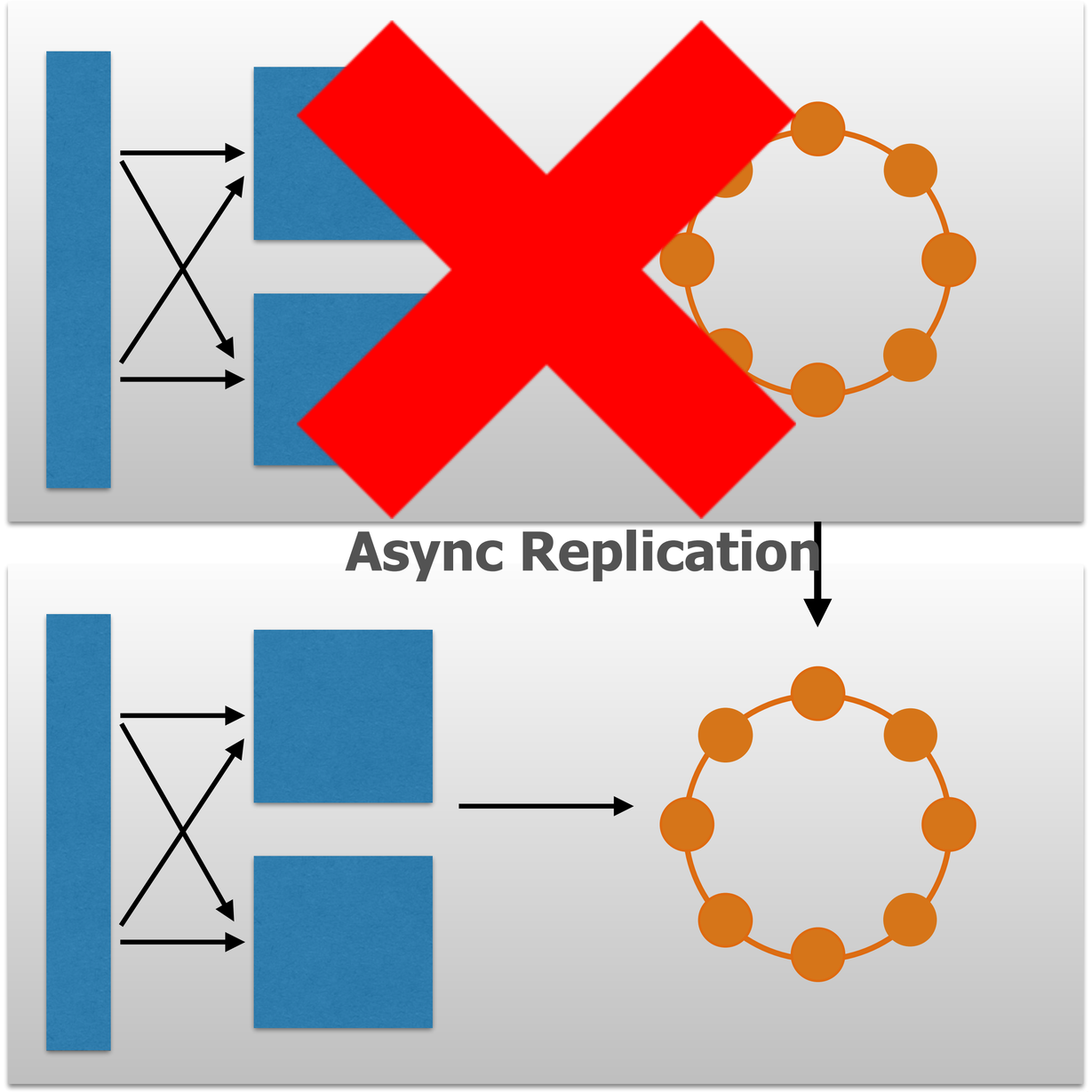
Spark with Cassandra



- Availability Model
  - Amazon Dynamo Paper
  - Distributed masterless
- Data Model
  - Google BigTable
  - Column family data model
- Multi data center replication
- Analytics with Apache Spark

Handling Failure





When Nodes Come Back?

- Hinted handoff to the rescue
- Coordinators keep writes for downed nodes for a configurable amount of time, default 3 hours
- Longer than that run a repair

I'M BACK
(DID YOU MISS ME?)


Gossip
Did you hear node 1 was down??
Consensus?

CQL - Easy

```
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;
```

CQL
Powerful & Expressive

```
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

Spark Cassandra Integration

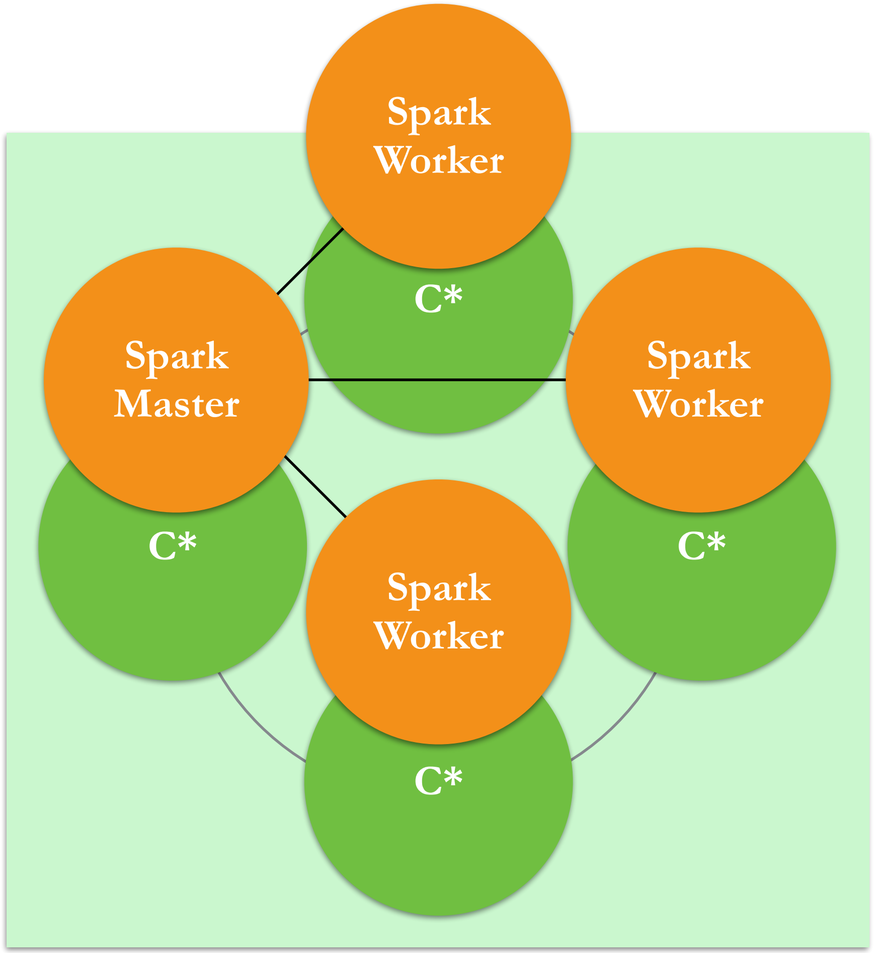
Spark Cassandra Connector

- Write data from Spark to Cassandra
- Read data from Cassandra to Spark
- Data Locality for Speed
- Easy and often implicit type conversions
- Offers an object mapper
- Server-Side Filtering
- Natural Timeseries Integration
- Implemented in Scala
- Also a Java API

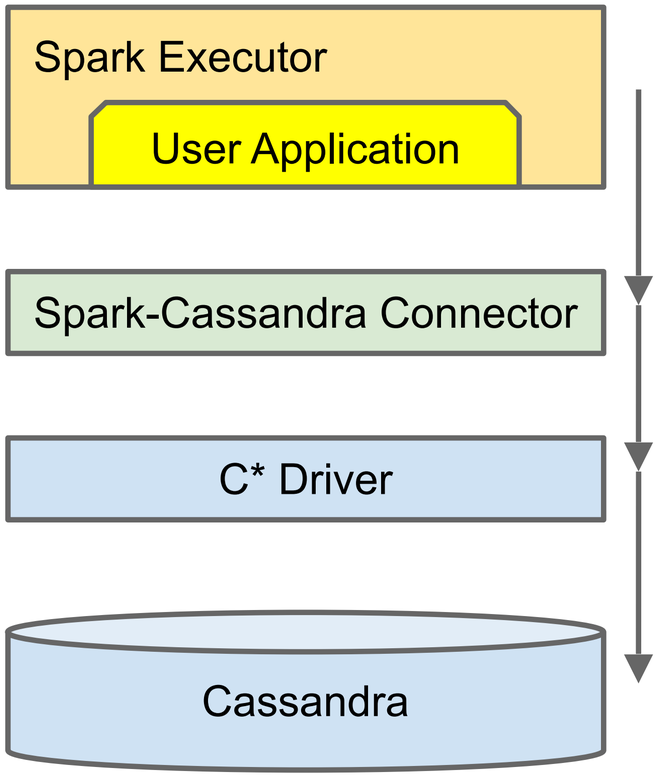
github.com/datastax/spark-cassandra-connector

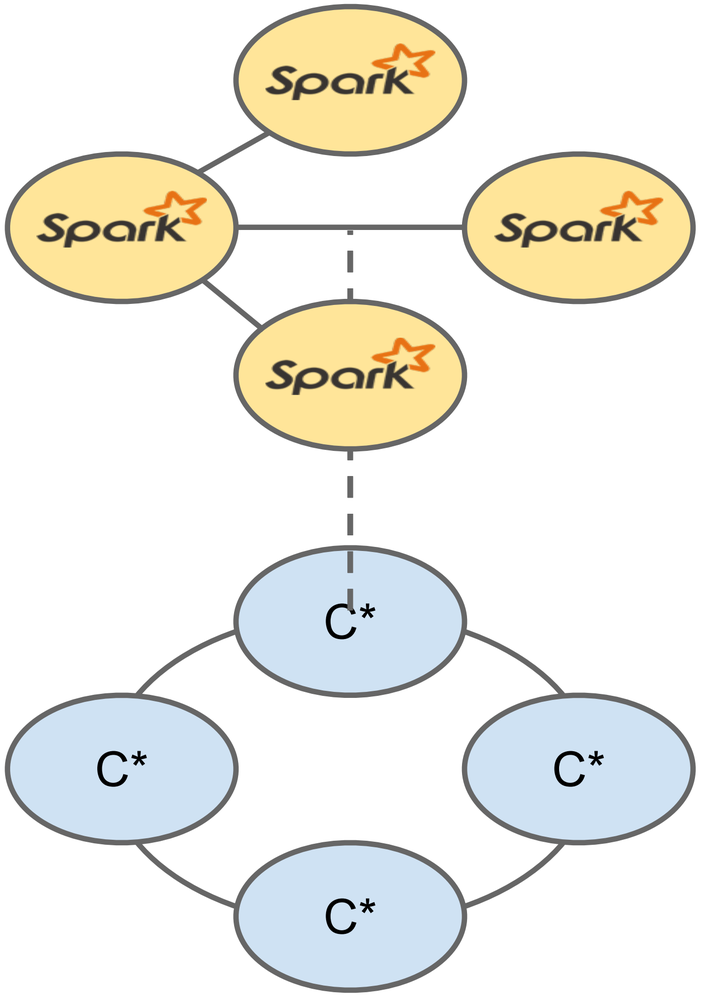
Co-locate Spark and Cassandra for Best Performance

Running Spark Workers on the same nodes as your C* Cluster saves network hops



Spark Cassandra Connector





Data Locality-Aware

Writing and Reading

SparkContext

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

StreamingContext

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

```
sparkConf.set("spark.cassandra.connection.host", "10.20.3.45")
```

Write From Spark to Cassandra

SparkContext

Keyspace

Table

```
sc.parallelize(collection).saveToCassandra("keyspace", "raw_data")
```

Spark RDD

JOIN with NOSQL!

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

Spark SQL with Cassandra

```
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, Cassandra & JSON

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

```
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}""")


val sql = new SQLContext(sparkContext)

sql.jsonRDD(json).map(CommitStats(_))
  .flatMap(compute)
  .saveToCassandra("github_stats","monthly_commits")
```

```
val rdd = sc.cassandraTable[MonthlyCommits]("github_stats","monthly_commits")
```

Cassandra, Kafka, Spark Streaming & 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)
```

Cassandra, Spark Streaming, Kafka

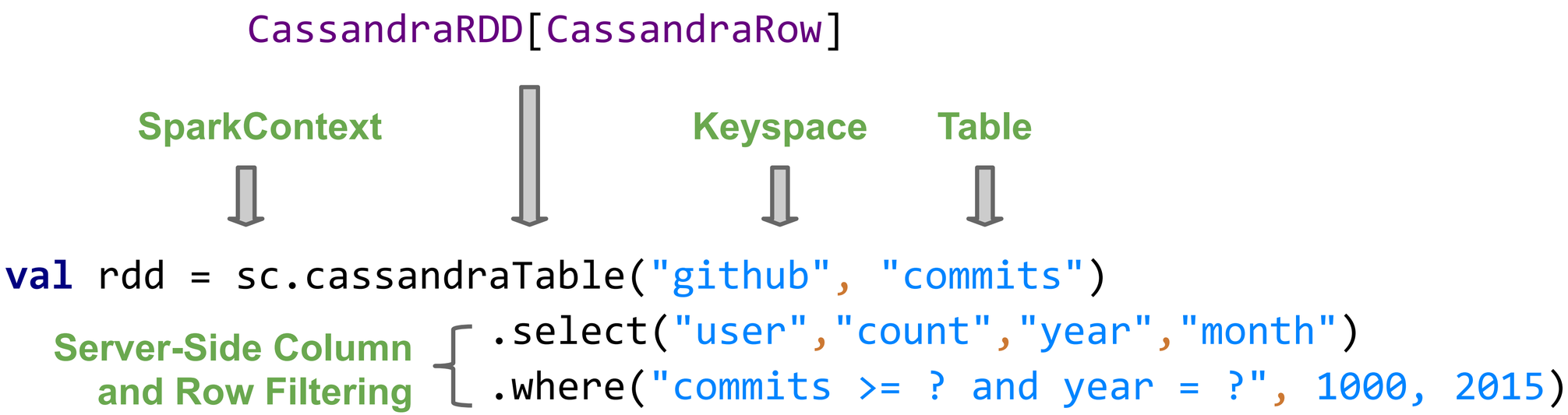
```
val streamingContext = new StreamingContext(sparkCtx, Seconds(30))


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

Cassandra, Spark Streaming, Twitter

```
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)
```

Reading: From C* to Spark



Rows: Custom Objects

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

```
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 keyValuesPairsRdd = sc.cassandraTable[(Key,Value)](keyspace, table)
```

Columns

```
val rdd: RDD[Song] = sc.cassandraTable[Song](keyspace, table).collect

val row: CassandraRow[Song] = rdd.first
val date: org.joda.time.DateTime = row.getDateTime("publication_date")

val list1 = row.get[Set[UUID]]("collection")
val list2 = row.get[List[UUID]]("collection")
val list3 = row.get[Vector[UUID]]("collection")

val nullable = row.get[Option[String]]("nullable_column")
```

Specify Rows

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

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

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

Working With Date Types

```
val rdd = ssc.cassandraTable("github", "c_aggregate")

rdd.where("time >= ?", "2014-07-12 20:00:02")

rdd.where("time >= ?", new org.joda.time.DateTime(2015,2,12,20,0,2)

rdd.where("time > ? and time < ?", "2014-07-12 20:00:01", "2015-03-18 20:00:03")
```

User Defined Types

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

*UDT = Your Field Type In C**

```
TypeConverter.registerConverter(new TypeConverter[MyUDT] {
  def targetTypeTag = typeTag[CustomerId]
  def convertPF = { case x: String => MyUDT(x) }
})

val udtRdd: RDD[(UUID, Int, MyUDT)] = sc.cassandraTable(keyspace, table)
```

UDT's With JSON

```
{
   "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)
);
```

Time Series

```
CREATE TABLE weather.raw_weather_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);
```

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

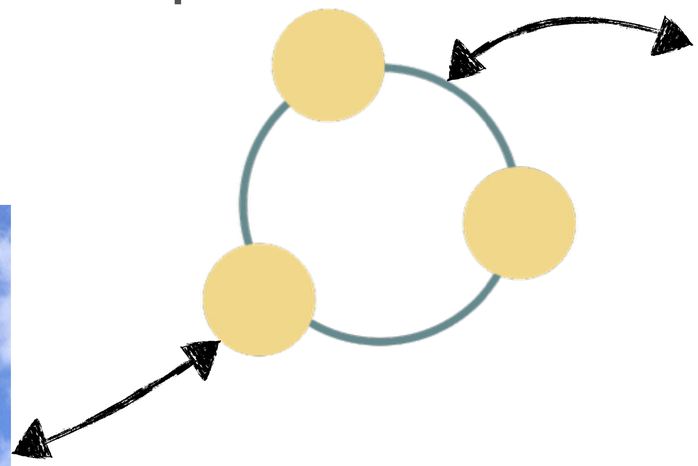
```
CREATE TABLE IF NOT EXISTS requests_ks.timeline (
  timesegment bigint, url text, t_uuid timeuuid, method text, headers map <text, text>, body text,
  PRIMARY KEY ((url, timesegment) , t_uuid)
);
```

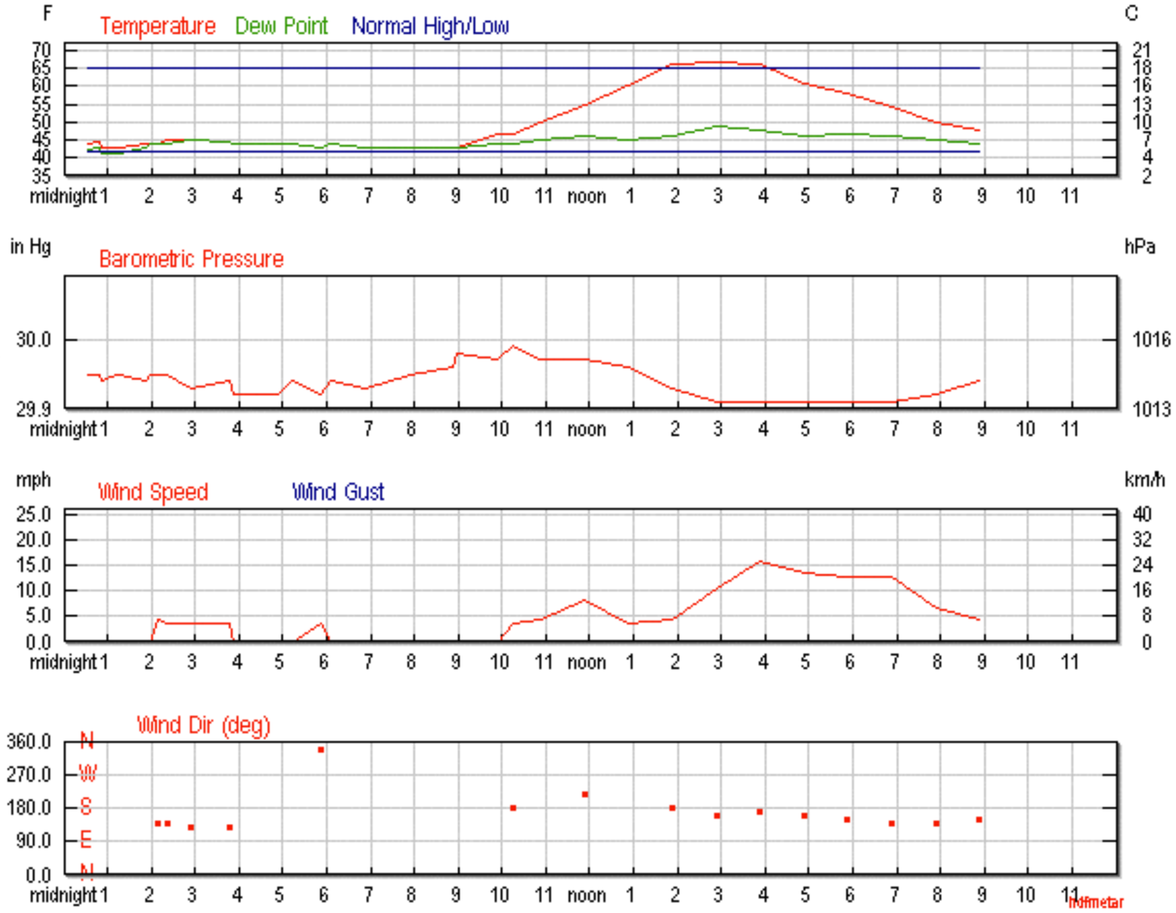
```
val multipleStreams = (1 to numDstreams).map { i =>
  streamingContext.receiverStream[HttpRequest](new HttpReceiver(port))
}

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

Application

Robust Application

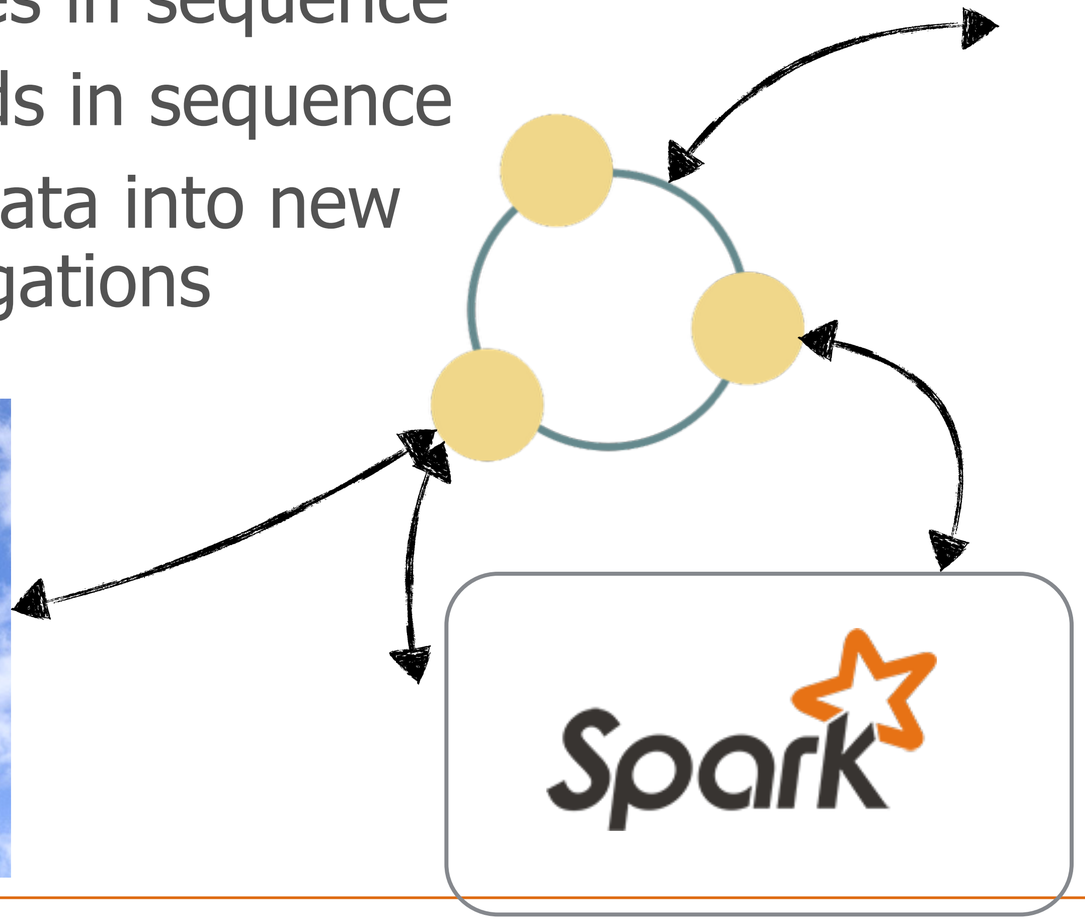
- Weather station collects data
- Cassandra stores in sequence
- Application reads in sequence





Weather Station Analysis

- Weather station collects data
- Cassandra stores in sequence
- Application reads in sequence
- Spark rolls up data into new tables of aggregations



Windsor
California

July 1, 2014

Data model should look like your queries

Queries I Need

- Get data by weather station
- Get data for a single date and time
- Get data for a range of dates and times
- Compute, store and retrieve daily, monthly, annual aggregations

Design Data Model to support queries

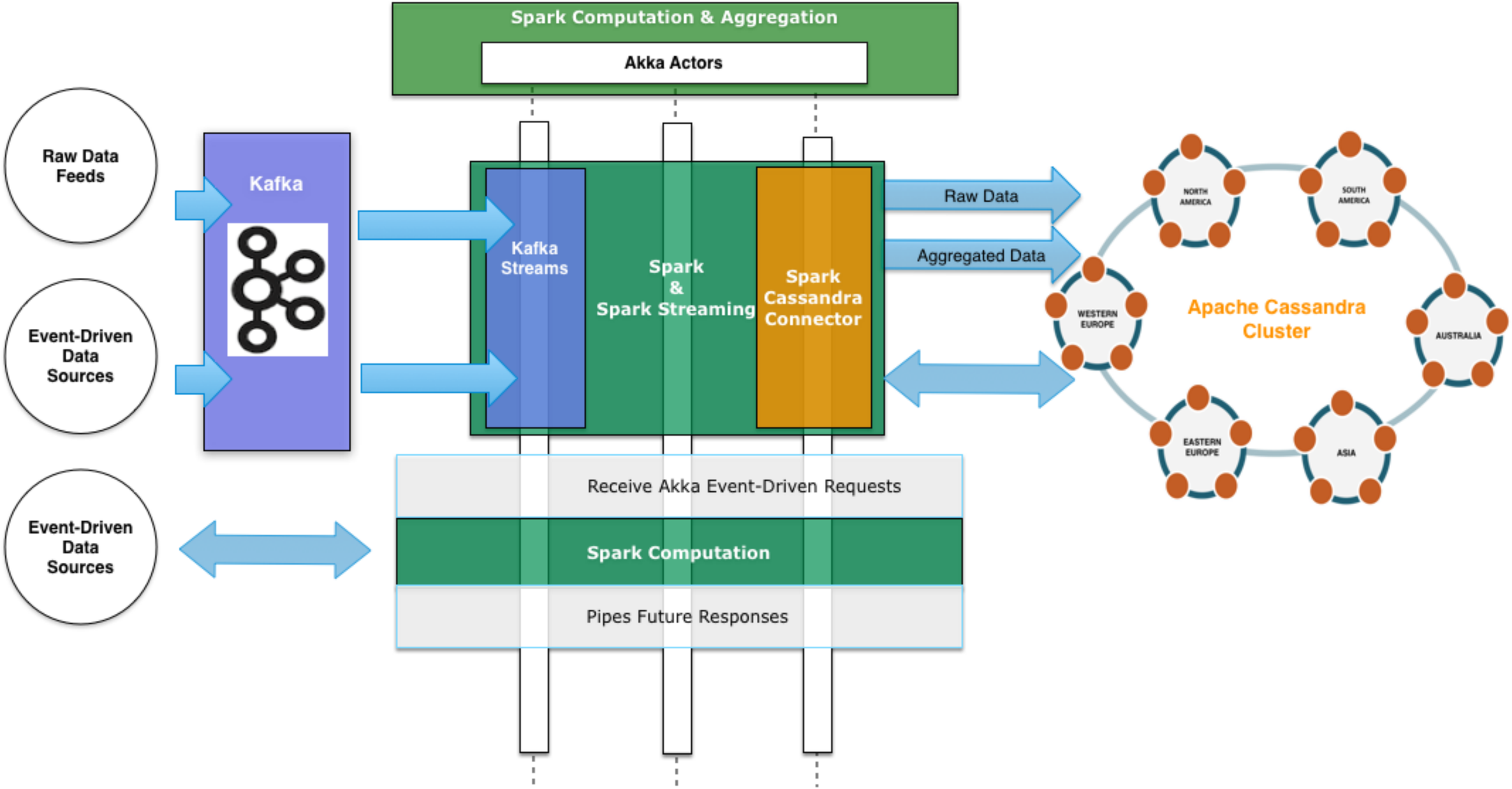
- Store raw data per weather station
- Store time series in order: most recent to oldest
- Compute and store aggregate data in the stream
- Set TTLs on historic data

Data Model

```
CREATE TABLE 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

Kafka Producer as Akka Actor

```
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)
  }
}
```

HTTP Receiver as Akka Actor

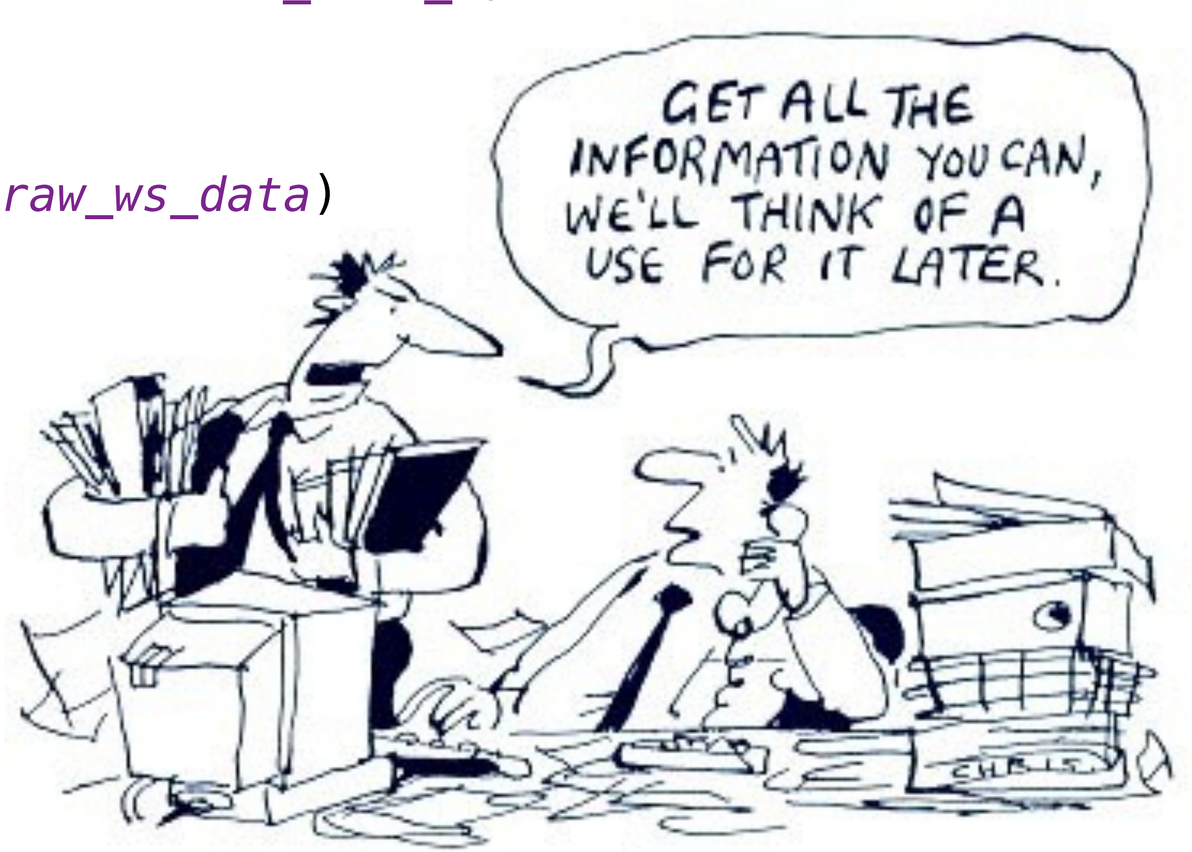
```
class HttpDataFeedActor(kafka: ActorRef) extends Actor with ActorLogging {
  implicit val materializer = FlowMaterializer()

  IO(Http) ! Http.Bind(HttpHost, HttpPort)

  val requestHandler: HttpRequest => HttpResponse = {
    case HttpRequest(POST, Uri.Path("/weather/data"), headers, entity, _) =>
      HttpSource.unapply(headers,entity).collect { case s: HeaderSource =>
        source.extract.foreach({ fs: FileSource =>
         kafka ! KafkaMessageEnvelope[String, String](topic, key, fs.data:_*)
        })
        HttpResponse(200, entity = HttpEntity(MediaTypes.`text/html`, s"POST successful."))
    }.getOrElse(HttpResponse(404, entity = s"Unsupported request"))
  }

  def receive : Actor.Receive = {
    case Http.ServerBinding(localAddress, stream) =>
      Source(stream).foreach({
       case Http.IncomingConnection(remoteAddress, requestProducer, responseConsumer) =>
         log.info("Accepted new connection from {}.", remoteAddress)
         Source(requestProducer).map(requestHandler).to(Sink(responseConsumer)).run()
   })
 }}
```

Akka: Load-Balanced Kafka Work

```
final class HttpNodeGuardian extends ClusterAwareNodeGuardianActor {

  val router = context.actorOf(
    BalancingPool(10).props(Props(
      new KafkaPublisherActor(KafkaHosts, KafkaBatchSendSize))))

  cluster registerOnMemberUp {

    val router = context.actorOf(BalancingPool(10).props(
      Props(new HttpDataFeedActor(router))), "dynamic-data-feed")

  }

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

Store Raw Data In The Stream First

```
val kafkaStream = KafkaUtils.createStream[String, String, StringDecoder,StringDecoder]
  (ssc, kafkaParams, topicMap, StorageLevel.DISK_ONLY_2)
  .map(_._2.split(","))
  .map(RawWeatherData(_))

kafkaStream.saveToCassandra(keyspace, raw_ws_data)
```



First Level of Aggregation

```
/** Per `wsid` and timestamp, aggregates hourly precipitation by day in the stream. */
kafkaStream.map { weather =>
  (weather.wsid, weather.year, weather.month, weather.day, weather.oneHourPrecip)
}.saveToCassandra(keyspace, daily_precipitation_aggregations)
```

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.

Second Level of Aggregation

```
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
- Batch and streaming in the same app

```scala
abstract class ClusterAware extends Actor with ActorLogging {

  val cluster = Cluster(context.system)

  override def preStart(): Unit = cluster.subscribe(self, classOf[ClusterDomainEvent])

  override def postStop(): Unit = cluster.unsubscribe(self)

  def receive : Actor.Receive = {
    case MemberUp(member) =>
      log.info("Member {} joined cluster.", member.address)
    case UnreachableMember(member) =>
      log.info("Member detected as unreachable: {}", member)
    case MemberRemoved(member, previousStatus) =>
      log.info("Member is Removed: {} after {}", member.address, previousStatus)
    case ClusterMetricsChanged(forNode) =>
      forNode collectFirst { case m if m.address == cluster.selfAddress =>
        log.debug("{}", filter(m.metrics))
      }
    case _: MemberEvent =>
  }
}
```

```scala
abstract class ClusterAwareNodeGuardian extends ClusterAware {
  import akka.actor.SupervisorStrategy._

  override val supervisorStrategy =   // customize
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1.minute) {
      case _: ActorInitializationException => Stop
      case _: IllegalArgumentException => Stop
      case _: IllegalStateException    => Restart
      case _: TimeoutException         => Escalate
      case _: Exception                => Escalate
    }

  override def preStart(): Unit = {
    super.preStart()
    log.info("Starting at {}", cluster.selfAddress)
  }

  override def postStop(): Unit = {
    super.postStop()
    cluster.leave(self.path.address)
    gracefulShutdown()
  }

  override def receive = uninitialized orElse initialized orElse super.receive

  def uninitialized: Actor.Receive = {
    case OutputStreamInitialized => initialize()
  }

  def gracefulShutdown(): Unit = {
    val timeout = Timeout(5.seconds)
    context.children foreach (gracefulStop(_, timeout.duration))
 }
}
```

```scala
class NodeGuardian(ssc: StreamingContext, kafka: EmbeddedKafka, settings: Settings)
  extends ClusterAwareNodeGuardian with AggregationActor with Assertions {
  import WeatherEvent._
  import settings._

  context.actorOf(Props(new KafkaStreamingActor(kafka.kafkaParams, ssc, settings, self)))

  val temperature = context.actorOf(Props(new TemperatureActor(ssc.sparkContext, settings)))
  val precipitation = context.actorOf(Props(new PrecipitationActor(ssc, settings)))
  val station = context.actorOf(Props(new WeatherStationActor(ssc.sparkContext, settings)))

  override def preStart(): Unit = {
    super.preStart()
    cluster.joinSeedNodes(Vector(cluster.selfAddress))
  }

  override def initialize(): Unit = {
    super.initialize()
    ssc.checkpoint(SparkCheckpointDir)
    ssc.start()
    context become initialized
  }

  def initialized: Actor.Receive = {
    case e: TemperatureRequest    => temperature forward e
    case e: PrecipitationRequest  => precipitation forward e
    case e: WeatherStationRequest => station forward e
  }
}
```

Roadmap

Recent Additions

Better write performance

- Token-aware writes
- Smarter batching
- Write throttling

Better read performance

- spanBy / spanByKey - timeseries data, better than groupBy
- Pushing down ORDER BY / LIMIT / COUNT to Cassandra

Recent Additions

- Scala 2.11 support and cross build
- Mapping user-defined classes to Cassandra UDTs
- Namespace support for multiple Cassandra clusters
- Transform RDD of PrimaryKeys into a CassandraRDD of another Table
- Spark SQL Integration Improvements
    - More predicate pushdowns
    - Support for joins across multiple clusters
- Metrics

Support For

- Push-down joins between generic RDD’s and C* Tables
- Partitioning any RDD to the same strategy as a C* Table
- Uses the source RDD's partitioning and placement for data locality

Roadmap

- CassandraInputDStream - stream from a cassandra table (soon)
- Performance Improvements
  - Token-aware data repartitioning
  - Token-aware saving
  - Wide-row support - no costly groupBy call
- Python API support
- Official Scala Driver for Cassandra
- Java 8 API

twitter.com/helenaedelson

github.com/helena

slideshare.net/helenaedelson

Resources

Spark Cassandra Connector

github.com/datastax/spark-cassandra-connector

github.com/killrweather/killrweather

groups.google.com/a/lists.datastax.com/forum/#!forum/spark-connector-user

Apache Spark spark.apache.org

Apache Cassandra cassandra.apache.org

Apache Kafka kafka.apache.org

Akka akka.io

Thanks for listening!

SEPTEMBER 22 - 24, 2015 | Santa Clara Convention Center, Santa Clara, CA

[Cassandra Summit]

Spark SUMMIT EAST